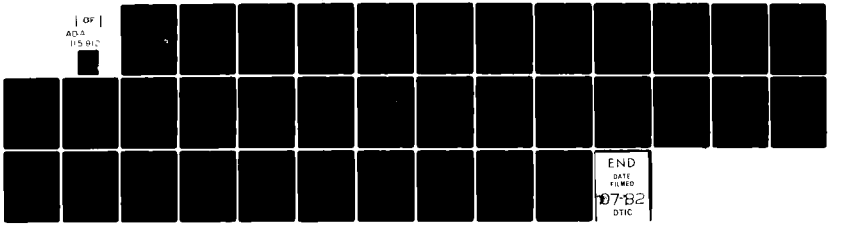
AD-A115 912 STANFORD UNIV CA CENTER FOR RELIABLE COMPUTING F/G 12/1
1982 INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING (FTCS---ETC(U)
APR 82 E J MCCLUSKEY DAAG29-82-K-0105
UNCLASSIFIED CRC-TR-82-3 ARO-18690.1-EL NL



Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ARO 18690.1-EL

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CRC Tech. Rpt. 82-3	2. GOVT ACCESSION NO.: AD-A115 912
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): 1982 INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING (FTCS-12) PREPRINTS (12)	5. TYPE OF REPORT & PERIOD COVERED: Interim Tech. Rpt.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Center for Reliable Computing, members of	8. CONTRACT OR GRANT NUMBER(s): ARO DAAG-29-82-K-0105
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Center for Reliable Computing, Computer Systems Laboratory, Stanford University; Stanford, CA 94305	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: DD Form 2222, Project No. P-18690-EL
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, Post Office Box 12211, Research Triangle Park NC 27709	12. REPORT DATE: April 1982
	13. NUMBER OF PAGES: 34
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Dr. William A. Sander, Electronics Division, U.S. Army Research Office 27709, P. O. Box 12211; Research Triangle Park, NC	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

N/A

18. SUPPLEMENTARY NOTES

THE VIEW, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHOR(S) AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Statistical failure models, workload, data analysis.

From A Statistical Load Dependency Model for CPU Errors at SLAC, page 6.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This technical report contains the text of Prof. McCluskey's keynote address and preprints of papers accepted for presentation at the 1982 International Symposium on Fault-Tolerant Computing (FTCS-12) which will be held in Santa Monica, California, June 21-24, 1982.

(over)

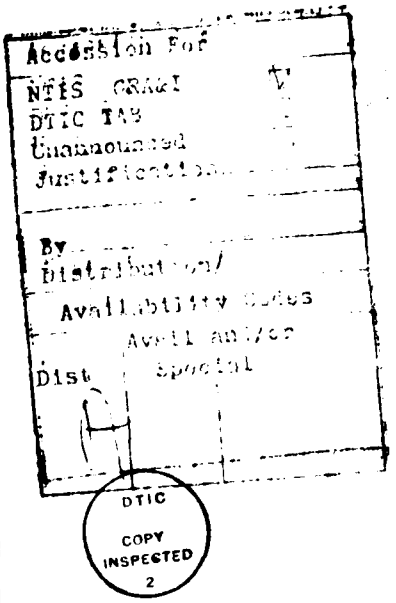
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

From A STATISTICAL LOAD DEPENDENCY MODEL FOR CPU ERRORS AT SLAC by Ravishankar K. Iyer and David J. Rosetti

This paper describes an analysis of CPU errors at the Stanford Linear Accelerator Center Computational Facility. The study includes all classes of temporary and permanent CPU errors. Nearly 85 percent of the errors are temporary failures. We find a strong load dependency in the errors. The observed tendency is present in three years of load data. This observation is significant because a load-failure relationship found at the CPU level must, in our view, be considered fundamental. In addition, the fact that most of the errors are transients or intermittents, provides new information on these error types with respect to their load dependent behavior. Our analysis procedure, used on the SLAC data, has been validated on an artificially created data base seeded with failures.



Center for Reliable Computing

1982 INTERNATIONAL SYMPOSIUM ON

FAULT-TOLERANT COMPUTING (FTCS-12) PREPRINTS

by the

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Depts. of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

CRC Technical Report No. 82-3

(CSL TN No. 200)

April 1982

These papers were supported in part by the National Science Foundation under Grant No. MCS-7904864, the U.S. Army Electronics Reseach and Development Command under Contract No. DAAK20-80-C-0266, the U.S. Army Research Office under Contract No. DAAG29-82-0105, the U.S. Dept. of Energy under Contract No. DE-AC03-76F00515, the Intel Corporation under the Honors Cooperative Program (HCP), and the Education Ministry of the People's Republic of China.

1982 INTERNATIONAL SYMPOSIUM ON

FAULT-TOLERANT COMPUTING (FTCS-12) PREPRINTS

by the

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Depts. of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

CRC Technical Report No. 82-3

(CSL TN No. 200)

April 1982

ABSTRACT

This technical report contains the text of Prof. McCluskey's keynote address and preprints of papers accepted for presentation at the 1982 International Symposium on Fault-Tolerat Computing (FTCS-12) which will be held in Santa Monica, California, June 21-24, 1982.

TABLE OF CONTENTS :

KEYNOTE ADDRESS

TEST QUESTIONS

E.J. McCluskey

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Stanford University, Stanford, California 94305

The program for FTCS-1 (1971) had 6 paper sessions and one panel session. The panel session was on diagnosis and testing. Two of the paper sessions involved testing: "Test Generation and Diagnosis" and "Fault-Location and Testing." Thus, over one third of the first symposium was devoted to testing issues.

There are 15 paper sessions, two panel sessions, and one keynote session at this symposium. Three of the paper sessions - "Design Testability," "Test Generation," and "Self-Test" are clearly devoted to testing topics. Another session, "On-Line Monitoring," is closely related and one-half of the papers in the session on "VLSI Design Issues" relate to testing. Somewhat less than 30% of this symposium is thus test-related. The attention given to testing hasn't changed very much from the first to the current FTCS Symposium.

Many conferences devoted entirely to testing have started since 1971: Cherry Hill Test Conference and Autotestcon are probably the most important of these. The conferences on testing typically cover very practical topics. They are organized and attended mainly by industry and government people. An exception is the annual Design for Testability Workshop, sponsored by the IEEE Test Technology Committee, which has a well balanced participation from academia as well as industry and government. In addition, testing papers have become common in many other conferences, most notably the Design Automation Conference.

Clearly the FTCS activity has not provided a sufficient vehicle to satisfy all of the current interest in testing. This is particularly evident by the fact that the IEEE Computer Society has started another Technical Committee - Test Technology - whose only topic is testing. Also another Technical Committee, the Computer Elements Committee, has now started an Annual Workshop on Testing. For someone like myself who has a major interest in testing it has become necessary to keep up with the activities of three technical committees as well as more than three annual conferences.

In 1971 the FTCS test papers were concentrated on the question of how to generate (minimum-length) test sets, and several of them presented sequential circuit test generation ideas. The emphasis has shifted significantly as evidenced by the present conference having sessions on: "Design for Testability," "Self-test," and "On-Line Monitoring," with only one session on "Test Generation." None of the papers appears to be specifically on sequential circuits although several address microprocessor testing.

In the 11 years between the first and the current conference, the complexity of digital logic has grown exponentially. Computer circuits have become ubiquitous in western society. The increased complexity has led to the realization that cost-effective automatic test pattern generation has become impossible for large designs that do not provide explicit testability-enhancing features. As a result, there is a great deal of interest in developing "Design for Testability" techniques.

In spite of much research, sequential circuit test generation is still extremely expensive. Adding scan path facilities to a design permits only combinational circuit test generation to be done. This technique is fast becoming standard in industrial and government designs. As complexity continues to increase, it is becoming evident that the cost of generating combinational circuit tests and applying them with a tester is starting to become too expensive. This has produced a great interest in the design of "Self-testing" circuits.

Although it is not illustrated by the program of this conference, another area of current concern is the question of the fault coverage obtained by the test technique used. With much denser chips two phenomena come into play: yield is lower and the chance of faults that are not adequately modeled as single stuck faults increases. These produce a requirement for higher fault coverage than was necessary in the past. The pervasiveness of digital technology has increased the need for some form of fault tolerance. In the test area this has caused increased attention to "On-Line monitoring" as well as increased test quality.

MODIFIED BERGER CODES FOR DETECTION OF UNIDIRECTIONAL ERRORS

Hao Dong

CENTER FOR RELIABLE COMPUTING, COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, California 94305, U.S.A.

ABSTRACT

Modified Berger codes are defined in this paper. They are less expensive than the ordinary Berger codes in terms of the number of check bits and the cost of checkers. As a trade-off, their error detection ability is slightly lower, although these codes can detect most unidirectional errors.

INTRODUCTION

It is seen that some physical defects in LSI or VLSI circuits tend to generate unidirectional errors. There are several classes of codes, such as m-out-of-n codes, Berger codes, and two-rail codes, that can be used to detect unidirectional errors. It has also been proved that Low-Cost AN Codes and Inverse Residue Codes with group length m+1 can detect all unidirectional errors of weight less than or equal to m [Wakerly 75], [Wakerly 78].

Berger codes have been proved to be the optimal separable codes that detect any unidirectional error [Berger 61] [Freiman 62]. However, in [Freiman 62], the author pointed out that one could not make a Berger code detect unidirectional errors of weight less than or equal to m by simply cutting down the number of the check bits, where m is an integer less than the number of information bits in a codeword. In this paper, we will define Modified Berger codes (MB codes) so that these codes will detect all the unidirectional errors of weight less than or equal to m. Then we will estimate the actual error detection ability of these codes. Totally self-checking checkers for MB codes are also described.

DEFINITION

A codeword of a Berger code has two parts: information D and check symbol C. Suppose D has I bits and C has k bits. Let I1 and I0 be the number of 1's and the number of 0's, respectively, in the I information bits. The check symbol of a codeword is the binary number of I0, or the complement (bit by bit) of the binary number of I1. That is

$$C = I0 \text{ or } C = (2^k-1)-I1.$$

We have

$$k=[\log_2(I+1)]$$

where [a] is the least integer greater than or equal to a. If $I=2^k-1$ then using I0 or I1 will result in the same code, and the code is called the Maximal Length Berger Code [Ashjaee 77].

Assume now all the erroneous bits are within the I information bits. If we use I0 modulo (m+1) or I1 modulo (m+1) (1<m<I) as the check symbol, denoted by C1, then all the unidirectional errors of weight less than or equal to m will be detected by this code, because no such error could change one codeword to another. In this case $J=[\log_2(m+1)]$ bits are needed for the check symbol C1. Let Pk (k=0, 1, ...) be the subset of codewords in which every codeword has an I1=k. The column C1 of Table 1 shows an example of such a code.

Table 1. A coding example for I=8 and m=7

Subset	Codeword example	I0	C1=I0 mod 8	C2
P0	00000000	8	000	111
P1	00000001	7	111	000
P2	00000011	6	110	001
P3	00000111	5	101	010
P4	00001111	4	100	011
P5	00011111	3	011	100
P6	00111111	2	010	101
P7	01111111	1	001	110
P8	11111111	0	000	111

A problem arises from the fact that the errors may also change the check bits. For example, an error may change a codeword in P1 to a codeword in P0 with only 4 erroneous bits (including the three check bits). As the number J usually is very small ($[\log_2(m+1)]$), it is reasonable to use a second level code to detect any error in the check bits. We may use any of the codes mentioned in the beginning of this paper to encode the check symbol C1 with another check symbol C2. These codes with check symbol C1 and C2 are called Modified Berger codes in this paper and the maximum weight of errors detected by an MB code is denoted by m. Table 1 shows an example of MB codes with m=7. The check symbols C1 and C2 in Table 1 form a two-rail code. In MB codes, because any unidirectional error in the check bits is detected by the second

coding, either I0 modulo (m+1) or I1 modulo (m+1) can be used directly as the check symbol C1. It is clear that the MB code in Table 1 (with check symbol C1 and C2) can detect any unidirectional error of weight less than or equal to 7, and that this error detection ability is effective regardless of the number of information bits in the code.

ERROR COVERAGE

From the definition of MB codes, we see that MB codes actually detect all unidirectional errors except those that affect only the information bits AND have weight equal to multiples of (m+1). In order to get an idea of the effectiveness of MB codes, here we give some estimation by two different error models.

Assume that the check symbol C1 is encoded in two-rail code by the check symbol C2. In this section we use the following notations:

- m: the maximum weight of unidirectional errors detected by the MB code.
- J: the number of bits in check symbol C1 or C2, $J=[\log_2(m+1)]$;
- I1: the number of 1's in the information bits;
- I0: the number of 0's in the information bits;
- I: the number of information bits, I=I1+I0;
- n: the length of a codeword, n=I+2J.

The total number of 1's (0's) in a codeword is I1+J (I0+J).

First, let us consider the independent error model. Under this model, an error on an output line is independent of the status of the other outputs. Suppose the probability that an error occurs on one output bit is p, and this probability is uniform for every output bit, and q=1-p. Then the probability that a unidirectional error occurs is

Prob (any unidirectional error)

$$= \sum_{i=1}^{I1+J} \binom{I1+J}{i} p^i q^{n-i} + \sum_{i=1}^{I0+J} \binom{I0+J}{i} p^i q^{n-i}$$

$$= (1-q^{I1+J})q^{I0+J} + (1-q^{I0+J})q^{I1+J}$$

$$= q^{I0+J} + q^{I1+J} - 2q^n$$

$$\doteq np \qquad (p \ll 1).$$

The probability that an undetected unidirectional error occurs is

Prob (undetected unidirectional error)

$$= \binom{I1}{m+1} p^{m+1} q^{n-(m+1)} + \binom{I1}{2(m+1)} p^{2(m+1)} q^{n-2(m+1)} + \cdots$$

$$+ \binom{I0}{m+1} p^{m+1} q^{n-(m+1)} + \binom{I0}{2(m+1)} p^{2(m+1)} q^{n-2(m+1)} + \cdots$$

$$\doteq [\binom{I1}{m+1} + \binom{I0}{m+1}] p^{m+1} \qquad (p \ll 1).$$

Then the conditional probability that a unidirectional error occurs but is not detected is

$$\frac{\text{Prob (undetected unidirectional error)}}{\text{Prob (any unidirectional error)}}$$

$$\doteq [\binom{I1}{m+1} + \binom{I0}{m+1})/n] p^m$$

This number will change from codeword to codeword, but essentially it should be very small for reasonable values of p and a value of m which is greater than 1. Also this probability will decrease exponentially when m increases. The reason for this is that the independent error model implies less probability for multiple errors. Although in some cases, such as a combinational circuit with fan-out points, the independent error model does not apply very well, in general, it is usually true that an error is less likely to occur if it involves more bits.

Next we consider another error model. Now we assume that all the unidirectional errors, no matter how many bits they affect, have the same probability to occur. Also assume that all the codewords have the same likelihood to be the output. The number of error patterns in a codeword is

$$\sum_{i=1}^{I1+J} \binom{I1+J}{i} + \sum_{i=1}^{I0+J} \binom{I0+J}{i}$$

$$= (2^{I1+J}-1) + (2^{I0+J}-1)$$

The number of codewords for I1 and I0 is

$$\binom{I}{I1} = \binom{I}{I0}$$

Let the total number of error patterns be E. We have

$$E = \sum_{I1=0}^{I} [(2^{I1+J}-1) + (2^{I0+J}-1)] \binom{I}{I1}$$

$$= 2 \sum_{I1=0}^{I} (2^{I1+J} - 1) \binom{I}{I1}$$

The number of undetected errors for each codeword is

$$\sum_{0<j(m+1)\le I1} \binom{I1}{j(m+1)} + \sum_{0<j(m+1)\le I0} \binom{I0}{j(m+1)}$$

Let the total number of undetected errors be D. Then

$$D = \sum_{I1=0}^{I} [\sum_{0<j(m+1)\le I1} \binom{I1}{j(m+1)} + \sum_{0<j(m+1)\le I0} \binom{I0}{j(m+1)}] \binom{I}{I1}$$

$$= 2 \sum_{I1=0}^{I} \sum_{0<j(m+1)\le I1} \binom{I1}{j(m+1)} \binom{I}{I1}$$

The error coverage of an MB code for all the unidirectional errors is 1-D/E. Table 2 shows this coverage for some MB codes in which C1 and C2 form a two-rail code. It is seen that when the number of information bits grows, the error coverage tends to stay around a fixed figure. This is a big advantage for those applications where the circuits have a large number of outputs. For a practical circuit, the error patterns that might occur will depend on the function and the structure of that circuit. In general, the error coverage of an MB code should be somewhere between the two models we analyzed. So we can say that MB codes will detect most of the unidirectional errors that may occur in a circuit.

Table 2. Comparison: Error coverage

I	MB Codes m+1	MB Codes 2J	MB Codes Coverage*	Berger Codes k	Berger Codes Coverage*
16	4	4	93.74%	5	100%
32	4	4	93.75%	6	100%
48	4	4	93.75%	6	100%
64	4	4	93.75%	7	100%
16	8	6	99.04%	5	100%
32	8	6	98.54%	6	100%
48	8	6	98.33%	6	100%
64	8	6	98.47%	7	100%

* Only for unidirectional errors.

CHECKING CIRCUITS

A general design procedure for Totally Self-Checking (TSC) checkers of Berger codes was presented in [Marouf 78]. The structure of these checkers is shown in Fig.1. In the diagram, circuit N1 is a weight generator which generates the weight of the information part D, that is, I1. Then the outputs of N1 are compared with the check symbol C by the comparator N2, which is implemented as a two-rail code checker. The weight generator N1 is a network of full adder (FA) and half adder (HA) modules. The procedures for constructing different weight generators are given in [Marouf 78]. This Berger code checker design can be easily modified for MB code checkers.

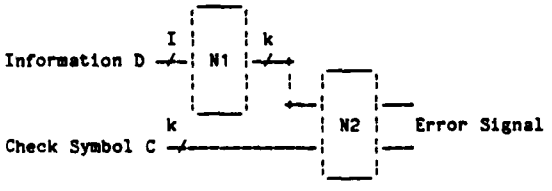


Figure 1. Structure of Berger code checkers

A TSC checker for an MB code consists of two parts as shown in Fig.2. Circuit CH1 checks the information bits by generating the complement of the check symbol C1 (by circuit N1') and comparing it with C1 (by circuit N2'). Assume that check symbol C1 is defined as

$$C1 = (2^J-1)-(I1 \text{ modulo } m+1).$$

In other words, C1 is the complement (bit by bit) of I1 modulo (m+1). In this case, circuit N1' generates the weight of information part D modulo (m+1). We call such a circuit N1' a modulo weight generator while the ordinary weight generators are refered to as full weight generators. Circuit N2' in Fig.2 is a two-rail code checker. The J outputs of circuit N1', denoted by C1', will then be compared with the check symbol C1 of the codeword by the checker N2'. Because MB codes provide the full code space for the two-rail code checker N2', the checking circuit CH1 described above is a TSC checker. The second level coding of C1 and C2 is checked by circuit CH2. C1 and C2 may form either a two-rail code or another Berger code. If C1 and C2 form a two-rail code, then CH2 is the same as N2'. When no error occurs, C1 and C2 are a codeword, so are C1 and C1'. We have C1'=C2, and f=f', g=g'.

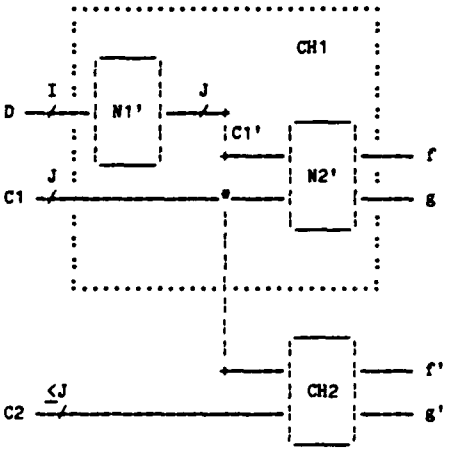


Figure 2. Structure of MB code checkers

Conceptually m+1 may be any integer, but in the case that $m+1=2^J$ the circuit implementation of the checker will be the simplest. In fact, a modulo weight generator can be obtained directly from the corresponding Berger code checker. This is done by keeping only the lowest J bits in each stage of weight representation and removing all the higher bits in the full weight generator. Fig.3 shows how a full weight generator with 15 information bits can be modified to realize a modulo weight generator with m+1=4 (modulo 4). In Fig.3, numbers for the full weight generator are noted in () if these numbers are different from those of the modulo weight generator. The asterisk (*) in an adder module indicates that for the modulo weight generator the adder does not have the highest carry output. So, for MB code checkers, the Jth bit of these adder modules is simply a three-input XOR gate instead of a full adder. It is also seen from Fig.3 that the modulo weight generator has less delay time than the full weight generator since its last adder module is one bit shorter.

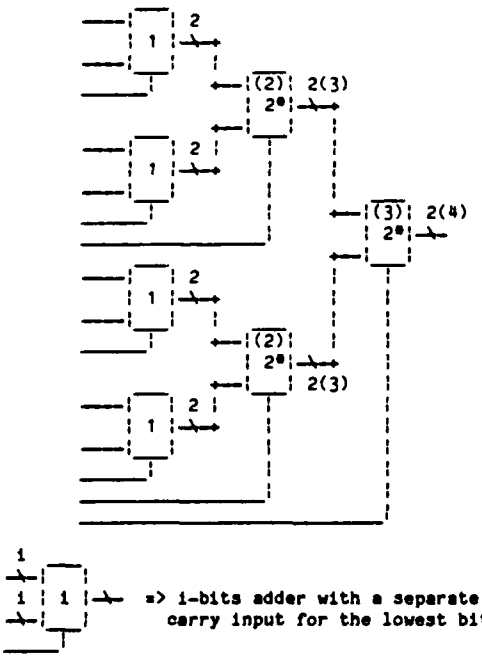


Figure 3. Modulo weight generator

Table 3 compares the hardware cost of check symbol generators of MB codes (m+1=4) with Berger codes and low-cost residue codes (group length=4). TSC checkers for low-cost residue codes are described in [Ashjaee 77] and [Avizienis 71]. The hardware savings in Table 3 are estimated in terms of the number of devices for PLA implementations and are compared with Berger codes.

Table 3. Comparison: Hardware cost for check symbol generators

Info bits I	Berger codes FA	Berger codes HA	LC codes FA	MB codes FA	MB codes HA	MB codes XOR*	Savings
15	11	0	-	7	0	3	22.7%
16	11	4	12	7	2	3	25.6%
31	26	0	-	15	0	7	28.8%
32	26	5	28	15	2	7	30.7%
63	57	0	-	31	0	15	32.5%
64	57	6	60	31	2	15	33.6%

* Three-input XOR gate.

CONCLUSION

Modified Berger Codes are defined in this paper. MB codes can detect all unidirectional errors of weight not equal to multiples of a predefined integer m+1. This error coverage is greater than low-cost residue codes but less than Berger codes. MB codes have fewer check bits than the corresponding low-cost codes or Berger codes in most cases. Also MB codes can be easily applied to any number of information bits. Because the number of check bits of an MB code is independent of the total number of the information bits, they are suitable for circuits that have a large number of outputs, such as PLA's. It is also shown in the paper that the totally self-checking checkers for MB codes are less expensive and have less time delay than that for either Berger codes or low-cost residue codes. All these advantages make MB codes very attractive for practical applications.

ACKNOWLEDGMENTS

This work was supported in part by a research fellowship given by the Education Ministry of the People's Republic of China and in part by the National Science Foundation under Grant MCS-7904864. The author wishes to thank Professor E. J. McCluskey and Professor J. F. Wakerly for their valuable comments.

REFERENCES

[Ashjaee 77] Ashjaee, M.J., and S.M.Reddy, "On Totally Self-Checking Checkers for Separable Codes", IEEE Trans. on Comp., Vol. C-26, pp.737-744, August 1977.

[Avizienis 71] Avizienis, A., "Arithmetic Codes: Cost and Effectiveness Studies for Application in Digital System Design", IEEE Trans. on Comp., Vol. C-20, pp.1322-1331, Nov., 1971.

[Berger 61] Berger, J.M.,"A Note on Error Detection Codes for Asymmetric Channels", Inform. Contr., Vol. 4, pp.68-73, March 1961.

[Freiman 62] Freiman, C.V., "Optimal Error Detection Codes for Completely Asymmetric Binary Channels", Inform. Contr., Vol. 5, pp.64-71, March 1962.

[Marouf 78] Marouf, M. A. and A.D.Friedman, "Design of Self-Checking Checkers for Berger Codes", Dig., 8th Ann. Symp. on Fault-Tolerant Computing (FTCS-8), June 21-23, 1978, Toulouse, France, pp.179-184.

[Wakerly 75] Wakerly, J.F., "Detection of Unidirectional Multiple Errors Using Low-Cost Arithmetic Codes", IEEE Trans. on Comp., Vol. C-24, pp.210-212, Feb., 1975.

[Wakerly 78] Wakerly, J.F., ERROR DETECTING CODES, SELF-CHECKING CIRCUITS AND APPLICATIONS, pp.40-50, Elsevier North-Holland, Inc., New York, New York, 1978.

A STATISTICAL LOAD DEPENDENCY MODEL FOR CPU ERRORS AT SLAC

Ravishankar K. Iyer and David J. Rossetti

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305 U.S.A.

ABSTRACT

This paper describes an analysis of CPU errors at the Stanford Linear Accelerator Center Computational Facility. The study includes all classes of temporary and permanent CPU errors. Nearly 85 percent of the errors are temporary failures. We find a strong load dependency in the errors. The observed tendency is present in three years of load data. This observation is significant because a load-failure relationship found at the CPU level must, in our view, be considered fundamental. In addition, the fact that most of the errors are transients or intermittents, provides new information on these error types with respect to their load dependent behavior. Our analysis procedure, used on the SLAC data, has been validated on an artificially created data base seeded with failures.

Keywords: Statistical failure models, workload, data analysis.

INTRODUCTION

It is well known that as a system approaches high levels of utilization, degradation in performance occurs [Ferrari 78]. An important question is whether increased system activity also results in the degradation of system reliability. If this is true, the implications are quite fundamental, since increased usage would result in an increased risk of error. Computing systems, which need maximum reliability at the time of their peak load, would require a reevaluation of their reliability projections. Research on the resolution of this question has been in progress at the Center for Reliable Computing at Stanford University since 1978. A lack of understanding of the complex physical interactions involved preclude analytical modeling at this stage. Accordingly, our approach has been to assume no model a priori, but rather start from a substantial body of empirical data on system load and failures. The object of the project is twofold:

1. To design and implement statistical experiments in an attempt to study the dependence of failure on load.

2. To develop models for determining any cause-effect relationships between workload and failures.

The techinques developed will form an important basis upon which analytical models and simulation techniques can subsequently be developed.

It is the purpose of this paper to report the results of our most recent investigations. These investigations were conducted on the IBM computer system at the Stanford Linear Accelerator Center (SLAC) computational facility. An overview of the SLAC system configuration appears in [Butner 80]. Using new techniques to measure both the workload and hardware errors in a large computer center for a period of three years, the following were completed:

1. The present study concentrates on CPU errors. A large majority of these can be classified as transient or intermittent.

2. We have now established a completely new data base of failures and load which is considerably superior to our old data base (UNILOG), [Butner 80] in depth, range and integrity. In particular, it captures a detailed internal view of the system and unlike UNILOG is automatically collected data.

3. More significantly, the workload and failure data were combined in order to match failures with workloads at the times of failure.

4. The measurements and statistical experiments clearly demonstrate an increased risk of CPU errors due to increased values of workload variables. Examples are CPU utilization, input/output rate, and interrupt rates.

A representative measurement is illustrated in Fig. 1, which shows how an increase in the input/output rate can result in higher risk of processor errors. The horizontal axis is the workload variable; the vertical axis is the risk of error. Modeling details will be given later in this paper.

Related Research

The failure data for initial studies, [Beaudry 78], [Butner 80] and [Iyer 81], came from the operator maintained data base called UNILOG. A statistical analysis of UNILOG failure data was performed in conjunction with a number of performance measures

from the IBM SMF[1] data log. In particular, we analyzed hardware and software failures, classified by component types. The study revealed a strong correlation between load and failures, although software failures correlated at a somewhat weaker level than hardware. Most importantly, the average overall system failure rate varied cyclicly over a band of significant width as determined by the daily load variations.

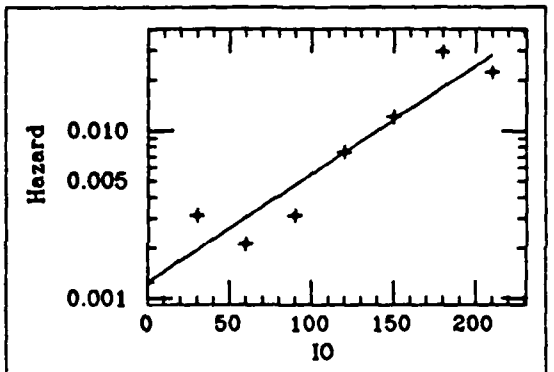


Figure 1: Risk of error increases with increasing I/O rate.

Additional substantiation of this result came from results reported in [Castillo 80], where a constant failure rate model is proposed. In experimenting with data from a DEC system, [Castillo 80] found a Poisson model to be valid only at specific hours of the day, for particular load levels. Subsequently, the same authors [Castillo 81] proposed the use of a doubly-stochastic Poisson process to model the cyclic load-failure relationship. The model assumes that the instantaneous failure rate can be described by a cyclostationary Gaussian process. In [Gunther 80] a novel theoretical model for an apparent dependency of failure on load, based on a random walk formulation, is described.

The next section motivates the current work and places our previous results in perspective. "Measurements" discusses the failure and workload measurements taken and briefly presents the organization of the data. Subsequent sections describe the analysis procedures and present new results. Finally, we summarize the important results and highlight the conclusions that can be drawn from them.

MOTIVATION

A user-oriented real time system is frequently expected to respond to conditions which differ from those for which it was modeled and evaluated. As indicated earlier, our approach has been to start with a substantial body of real data and examine it for a real or apparent dependency. In view of our previous results, we believe that the error process which ensues is composed of two separate effects. The first is the (constant) inherent failure rate. This is determined through classical reliability techniques [Shooman 68], taking into consideration such factors as topology, redundancy etc. The second is the utilization-induced failure rate. This rate is dependent upon both the absolute level of system utilization and the rate of change of that level. By an absolute level we mean an obviously measurable level; e.g., CPU utilization, memory occupancy, etc. Through the rate of change of utilization we are attempting to measure the rate at which transitions occur between various system states, e.g. the transitions of the CPU into and out of the busy state. Although the exact nature of these effects is not known, some underlying causes are thought to be as follows:

(i) Latent Discovery Effect: Many failures can only be detected when a particular module or subsystem is "exercised." In other words the system can be modeled as a load flow graph wherein we have increased path utilization when the load increases. Thus, although the failures may not be caused by increased utilization they are "revealed" by this factor. The time between the occurrence of failure and manifestation as a system error has been referred to as "error latency" [Shedletsky 73].

(ii) Increased Fatigue: There appears to exist a correlation between utilization and reliability. The more often we exercise information access channels and associated memory locations the greater the temperature and increase in fatigue.

(iii) Noise: A higher utilization level results in increased electronic noise. This can be expected to result in a higher error probability.

(iv) Synchronization and Timing Anomalies: The synchronization or timing anomaly category includes the failures due to time dependent aspects of the software and hardware. An error in the access to critical regions or an unanticipated sequence of status in an inter-computer communication protocol are some examples. Dependence upon level of utilization is obvious — as a system approaches full capacity, the "relative" timings of events can fluctuate widely. Sequences of events between processors can change from those originally anticipated as one or more of the CPUs nears saturation. A frequent source of timing anomaly is caused by implicit assumptions (often totally unintentional) regarding absolute times between events. In a real system pushed near 100 percent utilization, timings are highly dilated between system components and absolute synchronization assumptions can be violated.

Our previous studies did provide us with some insight into the above effects. We were, for exam-

[1] The IBM System Management Facilities, for collecting accounting and performance data. See [IBM 73] for details.

ple, able to see that though the latent discovery was an important factor, other effects were indeed present. The strong correlations in the pre-noon period (the "8 o'clock phenomenon") [Butner 80], shows that latent discovery is an important factor. The continued strong correlation in the afternoon suggests that other effects are also present.

We were, however, limited by the fact that the failure data was an external, human-collected view of the system. In order to obtain a closer insight into the problem, it was considered necessary to study the internal error generation process and determine its relationship, if any, with system activity. In particular, we decided to concentrate on CPU errors. An important reason for this was the fact that little is known regarding the behavior of the CPU errors and their relationship to load. In addition, a substantial number (85 percent) of the CPU errors, in the period of our study, were found to be "soft" errors, i.e. those from which the system recovered. Accordingly, they come in the general category of transient or intermittent errors (defined below), or design errors. Again, relatively little is known regarding the generation of these soft errors.

We define an intermittent error as one due to a component on the verge of failure. The error will re-occur frequently and eventually become permanent. It is generally believed that temporary failures are four to five times as frequent as permanent failures [Ball 69]. Nearly 90 percent of field errors are believed attributable to this class. Although a few analytical models exist, they are extremely restrictive and the basic assumptions need validation [Savir 77]. Statistical studies on real data are few and far between [McConnell 79]. The following section gives a general overview of the measurement techniques and construction of the data base.

MEASUREMENTS

Error Measurement

As stated earlier, the present study uses the most detailed data from the log maintained by the operating system as errors are detected by the hardware and recorded by the software.. High level system behavior, as seen by the computer operator and users, is not directly measured. Instead, there is much information on hardware errors, both permanent and non-permanent (transient and intermittent), as they occur in the detailed operation of system components.

The SLAC system, during the period of our study, consisted of two IBM 370/168 mainframes and an IBM 360/91 connected in a triplex mode. The data for our study, which consisted of three years of measurements (1979, 1980, and 1981), came from the two IBM 370/168 mainframes. The log referred to above is commonly called the "EREP" log, from the Environmental Recording Editing and Printing program used to accumulate and format it for maintenance [IBM 79]. Note that it is significantly more comprehensive than UNILOG which is essentially an external, human-collected log.

Errors in IBM 370 systems are classified into three major types:

1. CPU Errors - In the central processor and storage.

2. Channel Errors - In I/O channels and associated interfaces.

3. Outboard Errors - In any device beyond the channel-control unit interface, i.e. all errors in I/O devices.

For each error, whether recoverable or not, the operating system creates a time-stamped record describing the error and providing relevant information on the state of the machine. As an example, for a CPU error, the state information might include the contents of all internal registers and diagnostic information collected by the hardware (such as parity indicators and error flags).

At SLAC this information is collected on a daily basis and archived for many years. A small sample is presented in Table 1.

Workload Measurement

Since errors in processors occur fairly infrequently (on the order of once a day for our measurements), correlation with workload requires long term workload figures. Our workload data comes from two sources: the built-in system utilization facility, and a software monitor written specifically for this study. They are discussed below.

SMF Data. The operating systems in the processors measured use IBM's System Management Facilities (SMF) for usage accounting. SMF was originally designed to provide accounting information, but it has evolved over the years to include more general performance measurement information. SMF is discussed exhaustively elsewhere [IBM 73], [Butner 80] and will not be detailed here.

TABLE 1. Sample error data (EREP)

OBS	CPUMODEL	MCIC1	MCIC2	MCCOND	STRGBITS	SYSSTAT	DMGAREA	ERRORTYP	RMSACTN	RMSIHFO	PDARACTN	FAILTYPE	LOCATION	FOOTPRT1	FOOTPRT2	UTIME	SYSDMG	IPDMG	SYSREC	TIMDMG	EXTDMG	DEGRD	STRGERR	RECY	DMG	ERREXT	ERRSTOR	ERRECY	ERRCPU
35	168	04008F9F	00030588	04	8F	80	08	00	00	00	00	00	00	00	00	30MAR81:00:08:15					EDMG		UNCORR	HWRCY	PROC	Y	Y	Y	-
36	168	40008C9E	00030588	40	8C	20	80	A0	40	00	80	40	08	80	00	30MAR81:00:42:30		IDMG					UNCORR	TSKAB	PLSTR	-	Y	-	-
37	168	20008F9F	00030588	20	8F	20	80	A0	40	00	80	40	08	80	00	30MAR81:00:42:30			RCVY				UNCORR	TSKAB	RLSTR	-	Y	-	-
38	168	20008F9F	00030586	20	8F	80	08	00	00	00	00	00	00	00	00	30MAR81:00:42:33			RCVY				UNCORR	HWRCY	PROC	-	Y	Y	-

In general, SMF data consists of records giving resource utilization figures for jobs, files, I/O devices, and a potpourri of statistics gathered and written on a periodic basis. For this work we use the type 4 (Step) record, which holds statistics for each job step as it completes execution, and the type 1 (Wait) record, written roughly every 10 minutes, which summarizes global system utilization during that 10 minute period. With careful processing SMF can provide excellent workload statistics, especially when high resolution results are not needed.

INTRACK Monitor. To obtain more detailed information about transient behavior in the CPU we implemented an interrupt rate monitor, called INTRACK. This software monitor consists of two components: the interrupt counters and the INTRACK recorder. There are four classes of interrupts in the IBM 370 architecture:[2]

1. External (EXT) — Used by the operating system for clocks and inter-CPU communication.

2. Supervisor Call (SVC) — Caused by any SVC instruction. Used for operating system services, such as: memory allocation, synchronization, I/O, timing, etc.

3. Program (PROG) — Program traps due to arithmetic conditions (e.g. division by zero), invalid operations, or page faults.

4. Input/Output (I/O) — From completion of I/O operations.

The operating system provides an interrupt handler for each class of interrupt. A counter field and instruction to increment the counter were added at the beginning of each interrupt handler. These counters start at zero when the system is loaded and increase monotonically until the system crashes or is reloaded. The counters have the capacity to count up to 10^{15}, so overflow is not a problem.

The INTRACK recorder is a continuously running program that is automatically started every time the operating system is loaded. Table 2 summarizes the sources of data for our workload information. Figure 2 is an example of interrupt rates derived from the INTRACK counters.

The Data Base

Before the load and error data could be analyzed, it was necessary to create a coherent data-base which could be used as input in any subsequent analysis. This was particularly important for the workload data since the records came in varying formats and types. As a first step we created 5-minute time averages for all workload parameters for the entire period of our study.

[2] Machine check interrupts are not considered here because they are already collected in the EREP data.

TABLE 2

Input data for workload variables.

Record	When generated	Contents used
Step	At end of each batch job step	Accounting and job usage data, e.g. CPU time, No. of I/Os, memory usage.
Wait	Approx. every 10 minutes	CPU wait time during preceding 10 minute period.
INTRACK	Normally every 10 minutes (but settable)	Contents of four cumulative interrupt counters for: External, SVC, Program, I/O.

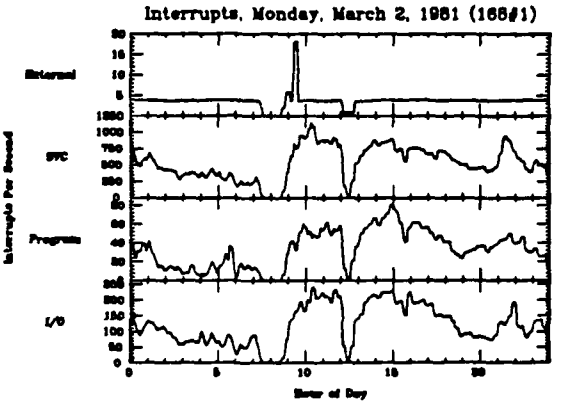


Figure 2: One day of INTRACK-collected interrupt rates.

In order to determine the load at the time of failure, the 5-minute load averages (which we refer to as smeared averages) were merged with the EREP log. The load at failure was taken to be the load in a five minute interval prior to the failure to eliminate perturbations from system error recovery or a system crash. The matching is shown in figure 3.

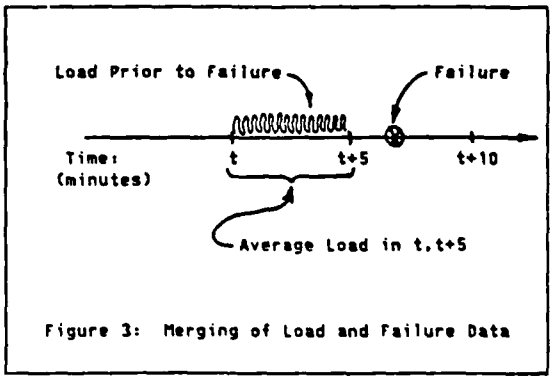


Figure 3: Merging of Load and Failure Data

As a second step, we also created an hourly smeared data base. The creation of these data bases necessitated complex processing in order to minimize the *loss of information* which invariably accompanies such procedures. The software system developed for this purpose is described in [Rossetti 81]. The system, which is highly interactive, allows efficient handling of large amounts of data (on the order of 4 x 10^9 bytes) of varying formats and complexities.

ANALYSIS

Initial Data Analysis

As an example of a simple analysis, let us summarize the types of machine checks that occur in the error data. We will count all unique patterns found in the machine check status bits provided by the hardware. The SAS[3] program used to generate Table 3 is less than fifteen lines long and was very simple to write. Each row in this table represents a unique pattern which may include one or more indicators that make up the error type. Positions containing "--" mean that the corresponding indicator was not in the pattern; abbreviations (such as SDMG, IDMG, etc.) are used as mnemonics for indicators that were set in the pattern. For example, the second row indicates that of the 456 errors occurring in the three years, 100 were hardware-recovered storage errors. The figures at the bottom of each column show the number and percent of errors for which the corresponding indicator was set.

TABLE 3

Breakdown of CPU Error Types

SYS DMG	INST DMG	SYS RCVY	EXT DMG	DE-GRADE	STOR-AGE	FREQ	PCT
--	--	--	EDMG	--	--	169	37.0
--	--	RCVY	--	--	STRG	100	21.9
--	IDMG	--	--	--	STRG	99	21.7
--	--	RCVY	--	--	--	46	10.0
--	--	--	EDMG	--	STRG	21	4.6
SDMG	--	--	--	--	--	11	2.4
--	--	RCVY	--	DEGR	--	6	1.3
--	--	RCVY	EDMG	--	STRG	1	0.2
--	IDMG	--	--	--	--	1	0.2
--	IDMG	--	EDMG	--	--	1	0.2
--	IDMG	--	EDMG	--	STRG	1	0.2
				Total Errors:		456	100.0
Totals for each indicator considered separately							
11	102	153	193	6	222	Frequency	
2%	22%	34%	42%	1%	49%	Pct. of All Errors	

A careful examination reveals much information about the types of processor errors and their relative severity. For example, external damage (EDMG) occurred in a large number of patterns (42%), and the table indicates that in almost all of those cases no other damage was detected in the CPU. *External damage is an error occurring in an area of* the system not directly connected with processing the current instruction. Another frequent category is system recovery (RCVY), at 34%, mostly in conjunction with some type of storage error (STRG). Apparently the system was able to recover by using error correcting codes or by retrying the instruction in progress. Notice that storage errors were involved in almost half the errors (222 or 49%) with about half of those (101 or 22%) being immediately corrected by the hardware. In fact, other tabulations show that the remaining storage errors were dealt with by operating system termination (54 or 12%), and task termination (76 or 17%). System damage (SDMG), which causes the operating system to stop immediately after recording the failure, occurred 2% of the time. The above shows that an assortment of fault recovery techniques are being used and contribute markedly to overall system performance. In fact, we find that in only 14% of the errors does the operating system stop processing.

Workload and Error Analysis

The data consisted of three years of load/failure measurements, 1979, 1980 and 1981. The 1981 data contains additional measurements made by our special purpose interrupt monitor. Initially, we analyzed each year separately. Since there was no significant difference in the 1979 and 1980 results, it was considered appropriate to combine the corresponding load-failure data. Of the thirteen workload measures collected for the study, four were chosen to be studied for 1979 and 1980. They were:

1. COREU — The sum of memory allocated by batch jobs (K bytes).
2. EXCP[4] — The I/O initiatation rate by batch jobs (I/Os per second).
3. SYSCPU — CPU utilization for system, i.e. non-batch, tasks (a fraction between 0 and 1).
4. TOTCPU — Total CPU usage (a fraction between 0 and 1).

For 1981 the following interrupt measurements were also included:

1. SVC — Supervisor calls (rate per second).
2. IO — I/O interrupts, completion of I/O operations (rate per second).
3. PROG — Program interrupts (rate per second).

The probability distribution $\mathcal{L}(x)$ of a workload variable is defined by

[3] The Statistical Analysis System is a powerful system for managing and analyzing data [SAS 79]. It was used for most of the data analysis.

[4] An acronym for "EXecute Channel Program"

$$\mathcal{L}(x) = Pr\ [workload = x]^{5}$$

and will be called the probability distribution of load. When failures are collected and matched to workload, the joint probability distribution of failure and load results, and is defined by

$$f(x) = Pr\ [failure\ occurs\ and\ load = x].$$

In this expression, failures and load values are represented as they occur on an actual system, where favored loads contribute more to the distribution than loads of low probability. To remove this effect we divide f(x) by the associated load probability $\mathcal{L}(x)$. Using the well known notion of a conditional probability distribution [Feller 68] we write

$$g(x) = Pr\ [failure\ occurs \mid load = x] = \frac{f(x)}{\mathcal{L}(x)}$$

Therefore g(x) can be thought of as the probability of a failure at a given load when all loads are equally represented: It is the conditional failure probability.

A commonplace analogy to illustrate the above distinction is that automobiles travelling at 150 mph have a higher probability of accident than those travelling at 55 mph. However, there are far more accidents for autos going 55. To obtain an accurate representation of the risks involved in travelling at high speed, we must divide the number of accidents occurring at each speed by the number of autos travelling at that speed. Figures 4 and 5 depict the $\mathcal{L}$, f, and g distributions of System CPU (SYSCPU) and Batch I/O Requests (EXCP) for 1979, 1980, and 1981.

THE LOAD HAZARD MODEL

In this section we describe and validate a model, hereafter referred to as a load-hazard model, which will form the basis of our tests for a possible load-failure dependency. It will be shown that if the load is acting as a stress on the system, then the load-hazard will increase with increasing load.

The object of our analysis was to determine whether a load-failure relationship exists in our data, i.e. whether a higher load stresses a system more than a lower load. In practical terms, if such an effect exists, we expect the load to act as a stress factor. The proposed model is similar in nature to the familiar hazard rate model from reliability theory. Recall that the hazard rate, which is the conditional probability that a system in operation at time t will fail in the interval (t,t+Δt), is defined in [Shooman 68] as:

$$z(t) = \frac{Pr\ [Failure\ in\ (t,t+\Delta t)]}{Pr\ [No\ failure\ in\ (0,t)]} \qquad (1)$$

A constant hazard rate implies that failures are occurring randomly in time, i.e. that there is an exponential failure relationship with time. An increasing hazard rate implies that the system is wearing out with time.

[5] The workload (or load) is assumed to be a discrete random variable for this discussion.

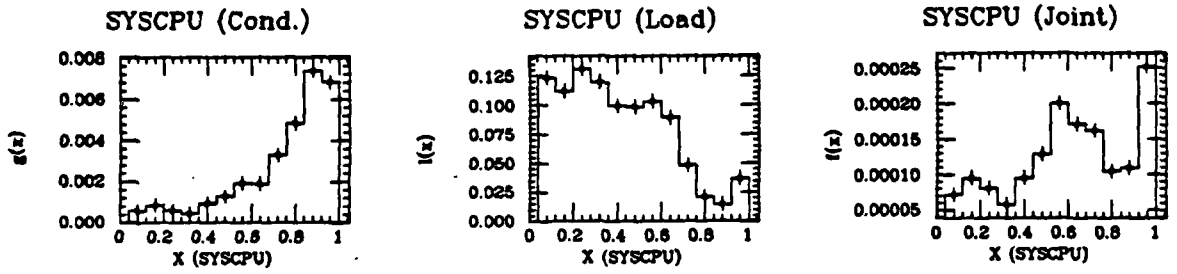


Figure 4. Frequency Distributions: System CPU - 1979, 1980.

System CPU is the fraction of CPU usage spent on system (i.e. non-user) tasks.

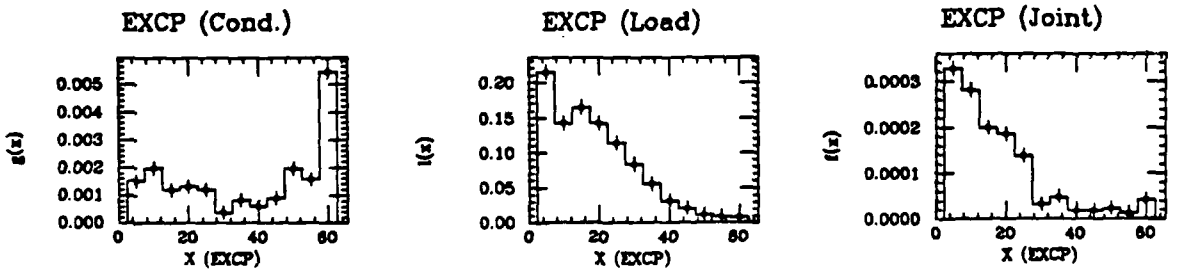


Figure 5. Frequency Distributions: EXCP - 1979, 1980.

EXCP is the number of user program I/O requests per second, based on five minute averages.

In close analogy with (1) above we propose a load dependent hazard. This is illustrated by the following elementary hypothetical experiment. Imagine that the system is operating in the range $0 < x < L$, where x is the actual system load and L its upper limit. Assume that we have M identical machines which are to be tested for a load-failure dependency. The experiment consists of testing each system for failures for increasing values of x. We commence by defining n increasing values of x, i.e. $x_1 < x_2 < \ldots < x_n$, at which we wish to test our computers. The machines are first run in the range $(0, x_1)$. The load on each machine is then increased from x_1 to x_2 and the number of failures are counted. The systems are then loaded from x_2 to x_3 and the failure frequencies established. This process is continued until the maximum load limit x_n is reached. If failures are load dependent, we expect that the risk of a failure will increase with increasing x in our experiment. This will be reflected in the corresponding frequencies. In more formal terms, we expect the probability that a system will fail at a load level $x \pm \Delta x$, given that it is currently running at x, will increase with increasing x.

The conditional probability described above bears a close resemblence to the classical hazard rate. Accordingly, we define a load hazard z(x) as

$$z(x) = \frac{\Pr\{\text{Failure in load interval } (x, x+\Delta x)\}}{\Pr\{\text{No failure in load interval } (0,x)\}} \qquad (2)$$

$$= \frac{g(x)}{1 - G(x)}$$

where: g(x) is the conditional failure probability,

G(x) is its cumulative distrib. function.

If z(x) increases with x, it should imply that the load is acting as a stress or wearout factor. If, however, z(x) remains constant for increasing x, we may surmise an exponential relationship with load.

Note that in our definition of load hazard we have removed the variability of system load by using g(x). Thus in the hypothetical experiment all loads are equally represented. This of course is not true in practice since load is best described as a random variable with a probability distribution; it is simply the associated load distribution, L(x), defined above. In order to determine the hazard for a particular load pattern, we must superimpose the associated load probability on the hazard calculated in (2). Denoting by $z_a(x)$ the transformed hazard, we have

$$z_a(x) = z(x)\, L(x) \qquad (3)$$

We refer to the hazard z(x), as defined in (2), as the fundamental hazard. This is because it can be thought of as an inherent property of a particular system and is not subject to varying load patterns. When a varying load pattern is taken into account, it can be thought of as "picking out" aspects of the fundamental hazard function. This hazard $z_a(x)$ defined in (3) will be referred to as the apparent hazard, since it is closely dependent on the load distribution.

Illustrative Example

The following example illustrates how a particular workload can modify a given fundamental load hazard z(x). Figure 6(a) shows a sample fundamental hazard z(x). Note that z(x) is increasing with load. Thus, if all load values are equally likely, the system has a higher risk of failure at higher load values than at lower load values. Fig. 6(b) is a hypothetical load distribution where the load variable is the fractional CPU utilization, with 0 for an idle CPU and 1 for a fully busy CPU. Finally, Fig. 6(c) gives the apparent hazard due to the effect of the load distribution in (a). The apparent hazard is now decreasing simply because higher load values are less probable.

Model Validation

Before using the proposed model on the SLAC load-failure data, we tested it on an artificially created data base. Our objective was to test if indeed the hazard model would predict a known dependency. Two tests were performed. In the first the load hazard was expected to remain unchanged with increasing load (i.e. that an exponential load-failure relationship exists). Thus:

$$\Pr\{\text{Load induced failure}\} = e^{-\lambda x}$$

where: x = system load

λ = constant load hazard parameter

A uniform load distribution was assumed. An artificial data base consisting of 20,000 load samples (5 minute averages) was created. The sample was

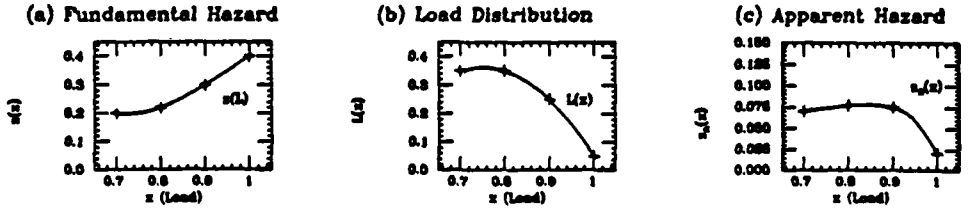


Figure 6: Example of Fundamental and Apparent Hazards

then seeded with failures, exponentially related to the load level (i.e. load to first failure is exponential). An unbounded arbitrary load parameter (e.g the I/O rate) was assumed. The failures were generated using an inverse transformation method similar to that described in [Fishman 73], for a hazard value of $\lambda = 0.001$. In the second test, the hazard was expected to increase with increasing load (e.g a uniform load failure relationship). A bounded load parameter (CPU usage) was modeled. In each case our hazard model was able to pick out the known dependency. The resulting fundamental hazards, as calculated by our formulation, are shown in figures 7 and 8.

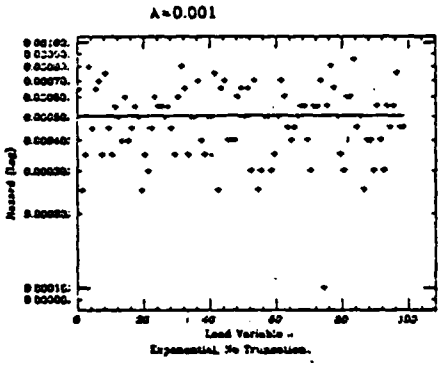


Figure 7: Hazard Plot: Exponential Model

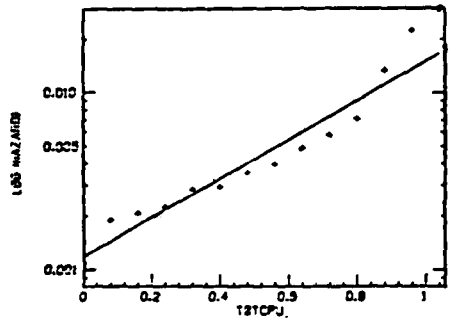


Figure 8: Hazard Plot: Uniform Model

HAZARD PLOTS

The generation of the hazard plots and associated statistics involved extensive data processing. In each hazard plot, $z(x)$ or $z_s(x)$ is calculated and plotted as a function of a chosen workload variable, x. In developing hazard plots for the load-failure data, there is an important difference between the real and the artificially created data. This lies in the fact that, while an artifical data base has specific dependencies seeded into it, in the real world, failures can occur due to a number of causes. Examples are: temperature, humidity, random noise, mechanical failures, and design errors, some of which are unrelated to our study. Those factors not related to load can be expected to behave as noise in a load-failure analysis. If these other factors are predominant, we can expect to find no discernable pattern in our hazard plots i.e. they should appear as uncorrelated clouds (e.g. see Fig. 9). This is well understood in any statistical study of dependencies.

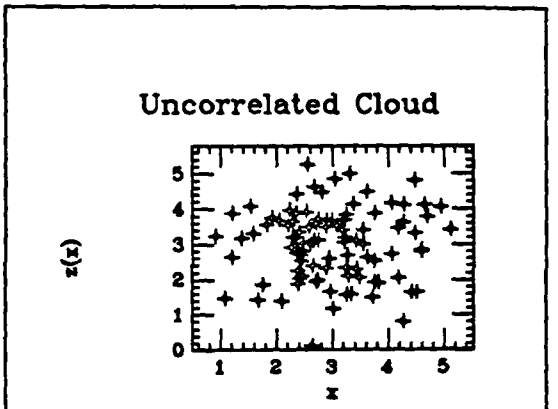


Figure 9: Uncorrelated Hazard Plot

An easily discernable pattern, on the other hand, would indicate that the load-failure dependency dominates others. The strength of such a relationship can be measured through regression. Figures 10, 11, and 12 depict the hazard plots for the three selected load parameters. The regression coefficient R^2, which is an effective measure of the goodness of fit, is provided for each plot. Quite simply, it measures the amount of variability in the data that can be accounted for by the regression model. R^2 values of greater than 0.6 (corresponding to an $R > 0.75$) are generally interpreted as strong relationships [Younger 79].[6]

[6] The range of $|R|$ from 0 to 1 is typically divided as follows: (0, 0.25) moderately weak; (0.25, 0.5) moderate; (0.5, 0.75) moderately strong; (0.75, 1.0) strong.

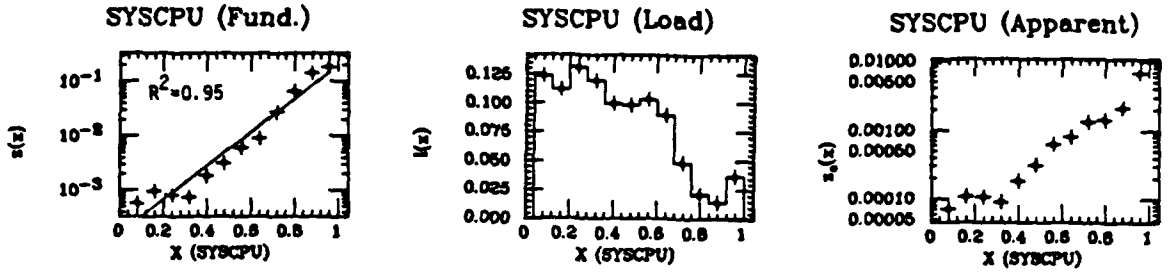


Figure 10: Hazard Plot: System CPU
The vertical scale is exponential in these plots, indicating that the hazard is rising sharply at peak loads.

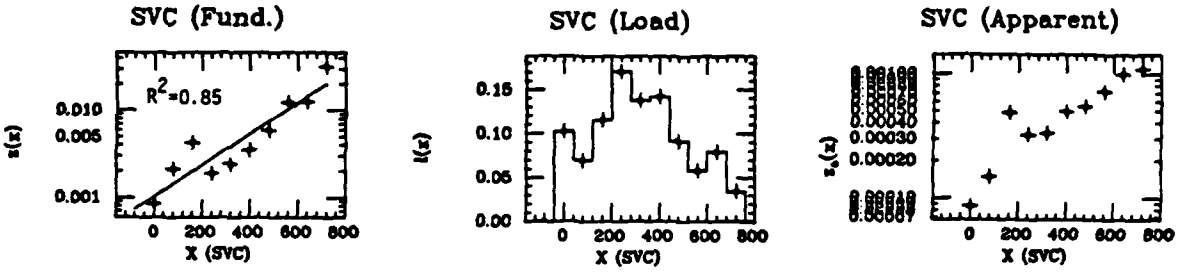


Figure 11: Hazard Plot: SVC

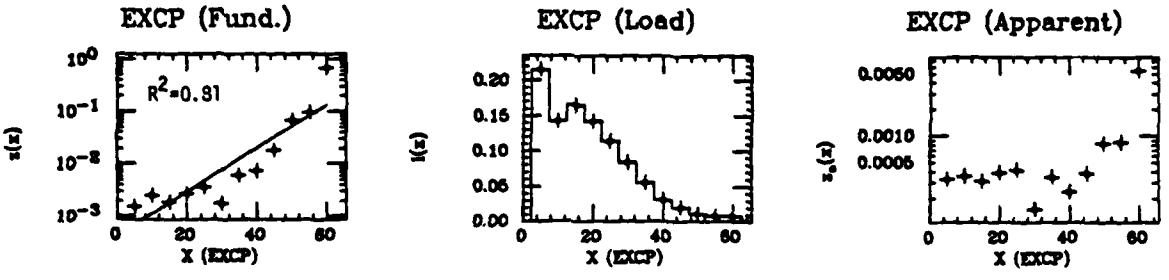


Figure 12: Hazard Plot: EXCP

It can be seen that the hazards are increasing with each of the load parameters shown. The relationship is particularly strong with system CPU or total CPU as load parameters. Thus it would appear from our data that the load parameters are acting as a stress factor, i.e. that there is an increasing risk of failure with increasing load.

Note, however, that there is some degree of overlap between the various load measures considered. Ideally, one would like to define and estimate a multivariate hazard function which correctly reflects the relative contribution of each load measure. In order to effectively achieve this goal it is necessary to construct a multivariate utilization function U (X_1, X_2, ... X_n) that relates the many varied measures of load to a single concept of sytem activity. It is expected that the function U would depend strongly on system configuration. The development of such a model is currently under investigation.

CONCLUSION

The analysis shows that there is a strong load dependency of internal CPU errors at SLAC. The observed tendency is present in three years of load data analyzed. This is significant because our previously reported results could only provide us with an external view of permanent system and component failures. By examining the CPU error generation process we have been able to study the inner behavior of the system and its reaction to errors. Consrquently, we have gathered the best data possible. A load-failure relationship found at this level must, in our view, be a fundamental phenomenon. In addition, the fact that a large majority of these errors are transients or intermittents provides new information on these error types viz. their load dependent behavior.

Our analysis procedure has been demonstrated on

artifically created data base seeded with failures. The two hazard models proposed clearly differentiate between fundamental (or inherent) and apparent load dependent failures. An estimate of the fundamental hazard z(x), provides the basic load-failure relationship. The apparent hazard $z_a(x)$ estimates how z(x) is modified by the load probabilities. It is, in principle, possible that even when no inherent relationship exists between load and failures, we could conceivably obtain an apparent dependency simply due to the fact that some load values occur more frequently than others. Alternatively, we can have the reverse situation where an increasing fundamental hazard is transformed into a non-increasing or even decreasing apparent hazard by a distinctive load distribution.

As with any statistical analysis, this is not proof in itself. However, the increasing body of evidence accumulated on different computers with differing load and failure patterns shows that workload should be considered as a factor relating to reliability. Workload can be thought of as a stress on the system, with greater stresses resulting in greater risk of failure. In most cases the effect of this stress is not permanent, since most errors are transient. The design of computer systems will be greatly aided if this type of analysis can help uncover cause and effect relationships in hardware errors.

ACKNOWLEDGMENTS

We gratefully acknowledge Prof. E.J. McCluskey for his continued interest in this work and for extensive discussions throughout the period of this study. We would also like to thank the SLAC computional facility, in particular, Ted Johnston for providing the computing resources and access to the data. In addition, we thank Prof. J.F. Wakerly and Dr. G. Rossmann for valuable discussions.

This work was supported in part by the National Science Foundation under Grant Number MCS-7904864, by the Department of Energy under Contract Number DE-AC03-76F00515, and by the Department of the Army under Contract Number DAAG29-82-0105.

REFERENCES

[Ball 69] M. Ball and F. Hardy, "Effects and Detection of Intermittent Failures in Digital Systems", 1969 FJCC, AFIPS Conference Proceedings, vol. 35, pp. 329-335.

[Beaudry 78] M. D. Beaudry "Performance Considerations for the Reliability Analysis of Computing Systems", Digest of Papers, Compcon, 1978.

[Butner 80] S. E. Butner and R. K. Iyer "A Statistical Study of Reliability and System Load at SLAC", Digest, Tenth International Symposium on Fault Tolerant Computing, October 1980.

[Castillo 80] X. Castillo and D. P. Siewiorek, "A Performance Reliability Model for Computing Systems", Digest, Tenth International Symposium on Fault Tolerant Computing, October 1980.

[Castillo 81] X. Castillo and D. P. Siewiorek, "Workload, Performance and Reliability of Digital Computing Systems," Digest, Eleventh International Symposium on Fault-Tolerant Computers, June, 1981, pp. 84-89.

[Feller 68] W. Feller, An Introduction to Probability Theory and its Applications, Wiley, 1968.

[Ferrari 78] D. Ferrari, Computer Systems Performance Evaluation, Prentice-Hall, 1978.

[Fishman 73] G. S. Fishman, Concepts and Methods in Discrete Event digital Simulation, Wiley, 1973.

[Gunther 80] N. L. Gunther and W. C. Carter, "Remarks on the Probability of Detecting Faults", Digest, Tenth International Symposium on Fault Tolerant Computing, October 1980.

[IBM 73] IBM Corp., OS/VS System Management Facilities (SMF), Order No. GC35-0004, 1973.

[IBM 79] IBM Corp., OS/VS, DOS/VSE, VM/370 Environmental Recording Editing and Printing (EREP) Program, Order No. GC28-0772, 1979.

[Iyer 81] R. K. Iyer, S. E. Butner, and E. J. McCluskey, "A Statistical Failure/Load Relationship: Results of a Multi-Computer Study," to appear in the IEEE Transactions on Computers, July 1982.

[McConnell 79] S. R. McConnell, D. P. Siewiorek and M. M. Tsao, "The Measurement and Analysis of Transient Errors in Digital Computing Systems," Digest, Ninth Annual International Symposium on Fault-Tolerant Computing, June 1979, pp. 67-70.

[Rossetti 81] D. J. Rossetti and R. K. Iyer, "A Software System for Reliability and Workload Analysis", CRC Tech. Rpt 81-18, Center for Reliable Computing, Computer Systems Laboratory, Stanford Univ., Stanford, CA. December, 1981.

[SAS 79] SAS Institute Inc., SAS User's Guide, 1979 Edition, 1979.

[Savir 77] J. Savir, "Model and Random Testing Properties of Intermittent Faults in Combinational Circuits", Tech. Note 118, Digital Systems Laboratory, Stanford Univ., Stanford CA., Aug 1977.

[Shedletsky 73] J. J. Shedletsky and E. J. McCluskey, "The Error Latency of a Fault in a Combinational Circuit", Digest, FTCS-3, June 1973.

[Shooman 68] M. L. Shooman, Probabilistic Reliability: An Engineering Approach, McGraw Hill, 1968.

[Younger 79] M. S. Younger, A Handbook for Linear Regression, Wadsworth Inc., 1979.

SELF-TESTING EMBEDDED PARITY TREES

Javad Khakbaz

CENTER FOR RELIABLE COMPUTING, COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, California 94305

ABSTRACT

This paper presents a procedure for modifying embedded parity trees so that they are tested by the inputs they receive during normal, fault-free, operation of the circuit. This eliminates the need for direct control over the input lines of the parity tree for testing purposes. The faults that are detected are single stuck-faults at the terminal lines of the XOR gates in the tree. Applications of this procedure to some other parity-related embedded code checkers are presented.

INTRODUCTION

A modular design for complex VLSI systems is necessary for many reasons. One such reason is "design for testability". It is simpler to deal with smaller blocks when the question of test pattern generation or error checking capability is addressed. Unfortunately, a system that consists of self-testing blocks is not necessarily self-testing. For example, consider a network that includes a combinational circuit B with inputs I_1 to I_p, outputs O_1 to O_4, and a parity tree C that calculates the parity of the outputs of B, as in Fig. 1. Parity tree C is tested for all single stuck-at faults at the terminals of the XOR gates by the test inputs shown in Table 1. However, suppose that by applying normal inputs to B, outputs of B receive only the patterns that are listed in Fig. 1. In this case, the network of Fig. 1 is not self-testing.

This simple example typifies the underlying problem in building a self-testing network by connecting self-testing blocks together. The problem is that it may be necessary, for applying test patterns, to have direct control over the input lines of an embedded block, i.e., a block some of whose input lines are not primary network inputs. Such direct control generally requires extra pins and/or circuitry on the chip and adds to the complexity of the design.

The above problem was recognized and explicitly considered by Anderson, [Anderson 71]. To build a self-testing network from self-testing blocks he required that each block be fully exercised, i.e., that it receive all its input codewords with the application of codewords to the inputs of the main network. This, however, poses a strong restriction on the design, and for some cases may be impossible to achieve. Smith defined the concept of sufficiently exercised blocks, which are self-testing (embedded) blocks that receive their test inputs during normal, fault-free, operation of the network, [Smith 76].

Based on Anderson's results, Wakerly concludes that the general problem of designing a network of

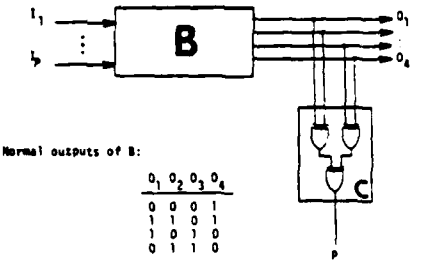


FIG. 1 An embedded parity tree.

O_1	O_2	O_3	O_4
0	0	0	0
1	0	1	1
1	1	0	1

TABLE 1 Test inputs for parity tree C of Fig. 1.

fully exercised (and, for that matter, sufficiently exercised) blocks is a difficult one to solve, [Wakerly 78]. Here, the discussion is limited to only one class of blocks, namely, parity checkers. A set of sufficient conditions are stated for the existence of sufficiently exercised embedded parity trees. If these conditions are satisfied, then a slight modification of the parity checker makes it such that the checker is tested by the input patterns that it receives during normal, fault-free, operation of the network. The faults that are detected by the normal inputs are the single stuck-at faults at the terminals of the XOR gates in the parity tree. The modification has no hardware cost or speed degradation associated with it.

NOTATIONS AND DEFINITIONS

Consider a network B^* and a combinational block B in B^*, as shown in Fig. 2. B has p input lines, I_1 to I_p, and n output lines, O_1 to O_n. The output of B is encoded using techniques such as even parity encoding. C is a checker that checks whether the output lines of B form a codeword. C must have at least two output lines, otherwise, its only output line may be stuck at its "good" logic value and this fault cannot be detected by applying codeword inputs to C. Therefore, as shown in Fig. 2, assume that C has two output lines. Usually the output lines of C form a 1-out-of-2 codeword. Thus the input of C is assumed to be correct if and only if the output lines of C carry complementary logic values. For more detail, see [Carter 68], [Anderson 71], and [Wakerly 78]. In this and the next section, assume that the input code space of C is the set of all n-bit words with even parity and the output code space of C is the set of 1-out-of-2 words. Since input lines of C come from the output lines of B, C is naturally an embedded block. Furthermore, the word patterns that the input lines of C can receive during normal, fault-free, operation of the network depend on the logic function of B, and in general are only a subset of all the even-parity n-bit words. The main objective of this work is to modify the parity checker C such that the normal inputs of C detect all single stuck faults at the terminals of the XOR gates in C.

Consider a Boolean matrix M whose rows are all of the (distinct) word patterns that the n output lines of B receive during normal, fault-free, operation of the network. If there are m such patterns, then matrix M is an m by n Boolean matrix. Note that all the rows of M have even parity. Call M the (normal) output matrix of B. The columns of M denote the logic values on individual output lines of B during normal, fault-free, network operation. Thus, there is a one-to-one correspondence between the columns of M and the output lines of B. The column in M that corresponds to output line O_i of B is called the

(normal) column corresponding to line O_i.

The traditional design of the single even parity checker C is obtained by partitioning the O_i lines into two arbitrary groups of preferably equal or almost equal sizes. Then, for each group, there is a parity tree that calculates the parity of its corresponding lines. The output of one of the trees is then inverted and, under normal conditions, forms a 1-out-of-2 code with the output of the other tree. For more detail, see, for example, [Wakerly 78]. Figure 3 shows an example of such design for 11 input lines. If the O_i lines (i.e., output lines of B) are connected to the input lines of C as shown in Fig. 3, then the normal column corresponding to the ith input of C (from the left) is the same as the normal column corresponding to O_i. However, note that any O_i can

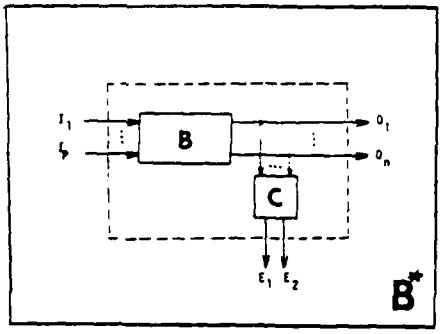
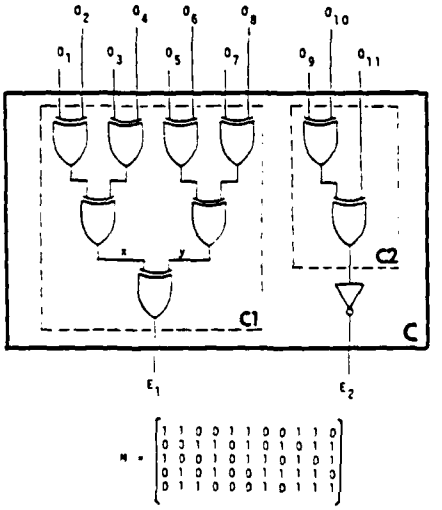


FIG. 2 The circuit under consideration.



$$M = \begin{bmatrix} 1&1&0&0&1&1&0&0&1&1&0 \\ 0&0&1&1&0&1&1&0&1&0&1 \\ 1&1&0&0&1&0&0&1&0&1&1 \\ 0&0&1&1&0&0&1&1&0&1&1 \\ 0&1&1&0&0&1&1&0&1&1&1 \end{bmatrix}$$

FIG. 3 An example.

be connected to any input line of C, as long as every O_i line is connected to exactly one input of C. This freedom is the basic tool utilized in the Algorithm A of the next section. The definition of normal columns can now be extended to apply to the internal lines of C. The normal column corresponding to any line x in C (Fig. 3) is an m-element Boolean column vector whose ith element is the logic value of line x when the output of B is the ith row of the normal output matrix of B, for i=1,2,...,m. That is, the components of the normal column corresponding to a line x are the logic values on line x for all normal outputs of B, applied in the order they appear in the normal output matrix of B. A line x in C is an active line if it receives both 0 and 1 during normal, fault-free, operation of the circuit. Thus, the normal column corresponding to an active line is a nonconstant column. A line with constant normal column is a passive line.

THE DESIGN OF SELF-TESTING EMBEDDED PARITY CHECKERS

For a given set F of possible faults in it, a parity checker C, as shown in Fig. 2, is said to be self-testing if for any fault f in F, there is a normal output pattern of B that causes either a <1,1> or a <0,0> output for C. Assume a design such as in Fig. 3 for C. Let the set F of faults consist of single stuck faults at the inputs and outputs of the XOR gates. Since there is always a sensitized path from any XOR gate terminal to the output of the parity tree, the embedded parity checker C is self-testing if and only every XOR gate terminal is an active line [Bossen 70]. Given a parity checker C and the normal output matrix for the block B, as exemplified in Figs. 2 and 3, Algorithm A below inspects every XOR gate terminal in C to see whether it is a passive line. If it is, then the Algorithm finds a new connection between the output lines of B and the input lines of C that makes that line active. After the termination of the Algorithm the connection prescribed by it makes every line in C active, and hence results in a self-testing embedded parity checker. In order for the Algorithm to work, the following conditions must be satisfied:

A1. Circuit C is implemented with two-input XOR gates.
A2. In M, the normal output matrix of B, no column is constant and no two columns are identical or complementary.

Two columns are complementary if they are complementary in all components. Note that Assumption A2 amounts to removing the redundant lines from the output of block B. If the above conditions are satisfied, Algorithm A below makes the embedded parity checker C self-testing. Note that in C (Fig. 3), any terminal of any XOR gate is the parity of a set of input lines of C. For line x, this set is denoted by S(x). Let M(x) be the binary matrix whose columns are the columns corresponding to the fault-free input lines in S(x). To check whether a line x is a passive line, that is, to check if the column corresponding to x is a constant column, one has to check whether the column obtained by XORing together the columns in M(x) is constant. In other words, x is passive if and only if the bit-by-bit XORing of the normal columns corresponding to input lines in S(x) results in a constant vector. Otherwise, x is active. In Algorithm A, E1 and E2 denote the two primary outputs of the parity checker.

Algorithm A:

1. If E1 is passive, exchange any arbitrary Oi in S(E1) with any arbitrary Oj in S(E2). (This exchange makes E1 and E2 active.)
2. Mark E1 and E2.
3. Consider input lines a and b of any XOR gate with marked output line and unmarked input lines. If, say, a is passive, exchange any Oi in S(a) with any Oj in S(b). If this exchange makes b passive, exchange Oi, which is now in S(b), with Ok, a member of S(a) different from Oj. (If before Step 3 both a and b are not active, then this Step makes them active in at most two exchanges.)
4. Mark a and b.
5. If there are no more unmarked lines, EXIT; otherwise, go to 3.

In the Appendix it is proved that if conditions A1 and A2 are satisfied, then Algorithm A makes the parity checker C self-testing.

EXAMPLE: Once again, consider the example of Fig. 3, with the specified normal output matrix for B. Since E_1 corresponds to the parity of the first 8 inputs of C, with the connection shown in Fig. 3, E_1 will have the following normal column:

0
0
0
0
1

So, mark both E_1 and E_2. Since E_1 is marked, consider lines x and y. Line x is the parity of the first four input lines of C. Therefore, the normal column corresponding to x is an all-0 column; i.e., x is a passive line. To make x active, exchange the connections of O_4, which is in S(x), and O_5, which is in S(y). This results in matrix M' of Table 2, which is obtained by exchanging columns 4 and 5 of matrix M. This exchange makes x active; however, line y becomes passive, as its corresponding normal column is now an all-1 column. For this case, Algorithm A cancels the latest exchange, and instead exchanges O_5 with another member of S(x), say, O_3. This exchange results in matrix M" of Table 2, which is obtained from M by exchanging columns 3 and 5. It makes both x and y active. The continuation of the Algorithm results in no more exchanges. The matrix M" is translated into the connection shown in Fig. 4. All the lines in parity checker C of Fig. 4 are

active. Therefore, the embedded checker C is self-testing.

$$M' = \begin{bmatrix} 1&1&0&1&0&1&0&0&1&1&0 \\ 0&0&1&0&1&1&0&1&0&1&1 \\ 1&0&0&0&1&1&1&0&1&0&1 \\ 0&1&0&0&1&0&1&1&1&1&0 \\ 0&1&1&0&0&0&1&0&1&1&1 \end{bmatrix} \qquad M'' = \begin{bmatrix} 1&1&1&0&0&1&0&0&1&1&0 \\ 0&0&0&1&1&1&0&1&0&1&1 \\ 1&0&0&1&0&1&1&0&1&0&1 \\ 0&1&0&1&0&0&1&1&1&1&0 \\ 0&1&0&0&1&0&1&0&1&1&1 \end{bmatrix}$$

TABLE 2 Modified output matrices for the example of Fig. 3.

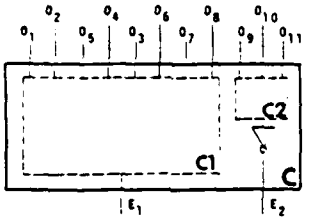


FIG. 4 Self-testing connection for the parity checker of Fig. 3.

APPLICATIONS

Algorithm A can be used to design self-testing embedded checkers for other parity-related encoding schemes.

Self-Testing Embedded Two-Rail Checkers

The self-testing two-rail checker tree with n input pairs, as described in [Carter 68] and [Anderson 71], has a one-to-one correspondence with an n-input parity tree, where each input of the parity tree is replaced with an input pair from the two-rail code, and each XOR gate is replaced with a two-rail checker with two input pairs and a 1-out-of-2 output code. Fig. 5 shows a self-testing two-rail checker tree with 8 input pairs. The corresponding parity tree for this is shown in Fig. 6. If the O_i lines of Fig. 5 satisfy Assumption A2, then Algorithm A can be applied to the circuit of Fig. 6, and any changes done on this circuit can readily be translated back into the original two-rail checker of Fig. 5. If line T of Fig. 5 (and hence of Fig.6) is passive, the two-rail input pairs should be partitioned into two arbitrary groups, as in self-testing parity checker design, and the pairs in each group should have a separate two-rail checker. A trivial such partitioning for circuit of Fig. 5 is shown in Fig. 7. As far as speed is concerned, this is not a good partition; however, other partitions are possible that result in faster checkers. Now each tree in Fig. 7 can be translated into a parity tree, as described above.

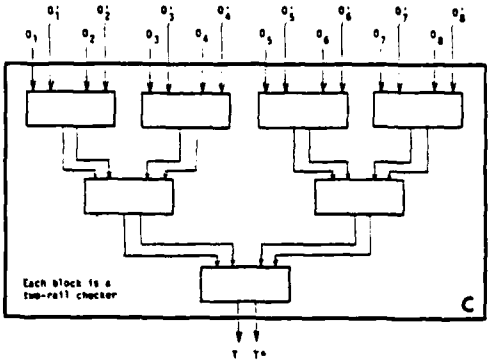


FIG. 5 A self-testing two-rail checker tree.

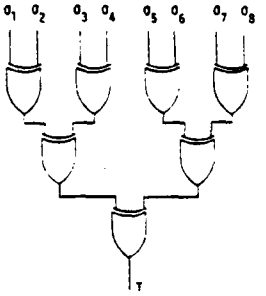


FIG. 6 Parity-tree equivalent of Fig. 5.

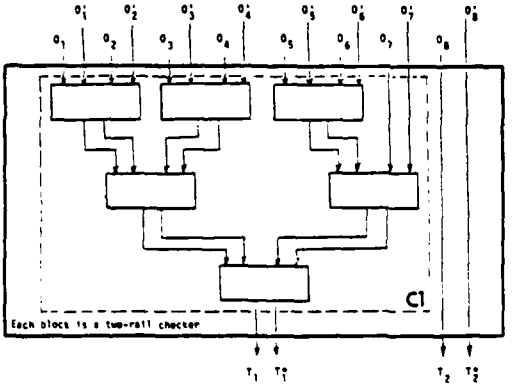


FIG. 7 A partitioned two-rail checker.

Self-Testing Embedded Detector for SEC-DED Circuits

Single error correcting and double error detecting (SEC-DED) codes are a very popular means of checking/correcting faults in memory arrays. The general scheme for SEC-DED decoders is shown in Fig. 8. Such circuits are naturally embedded. In particular, if they are used with ROMs, one may not be able to apply the required test patterns to these circuits since the contents of ROMs are predetermined. Thus it is necessary to make modifications to make such embedded SEC-DED circuits self-testing. Here we use Algorithm A to make the detector portion of the circuit self-testing.

As a specific example, consider 16-bit input data. This requires 6 check bits. Thus, in Fig. 8, n=16 and m=6. Hsiao has provided an optimal circuit for this case, [Hsiao 70]. The same design has been used in some commercial products, e.g., TI's SN54/74LS630. Let the data lines be denoted by d_0 to d_{15}, and let the check bits be c_1 to c_6.

Figures 9 and 10 show the design of the syndrome generator and the error detector, respectively, as given in [Hsiao 70]. The control lines have been left out for simplicity. The following describes a procedure for making the the detector portion of the circuit of Fig. 8 (i.e., circuits of Figs. 9 and 10) self-testing. The procedure works if all the 22 input lines of the SEC-DED circuit satisfy Assumption A2.

First, each XOR tree block of the syndrome generator must be modified as shown in Fig. 11. For the particular example at hand, there are six such parity checkers, each with a 1-out-of-2 output code. Thus the output of the syndrome generator is a two-rail code with six pairs, $\langle E_1, E_1' \rangle$ to $\langle E_6, E_6' \rangle$. If for all normal inputs to the circuit $E_1 \oplus \ldots \oplus E_6$ is constant, take any one of the six parity checkers of the syndrome generator and exchange line O_9 with any of the other eight lines, (Fig. 11). After this, the above parity is no longer constant [Khakbaz 82a]. For any of the six parity checkers just obtained, use algorithm A to make it self-testing. Since all the E_i' output lines of the syndrome generator are (inverted) primary inputs to the SEC-DED circuit, they satisfy Assumption A2. Hence an (embedded) self-testing two-rail checker can be designed, as described above. This replaces the OR gate of Fig. 10.

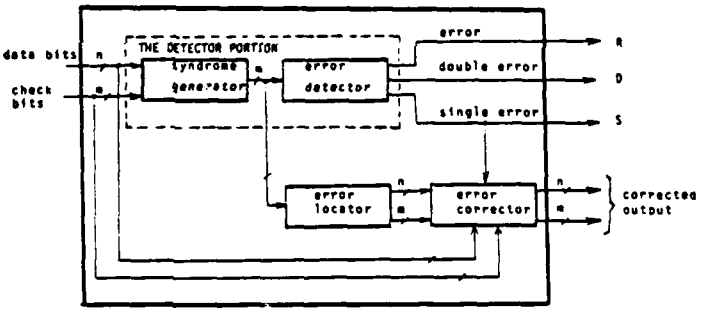


FIG. 8 SEC-DED decoder circuit.

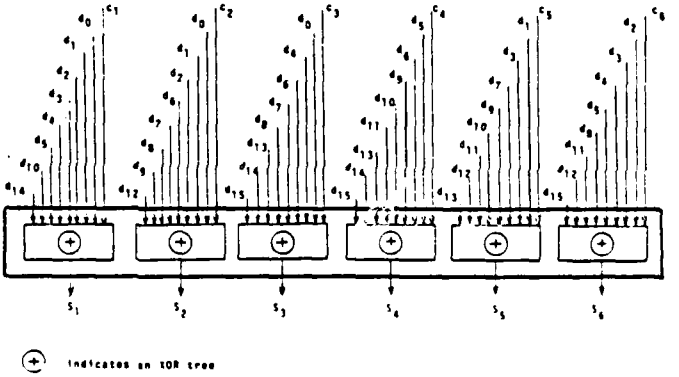


FIG. 9 The syndrome generator.

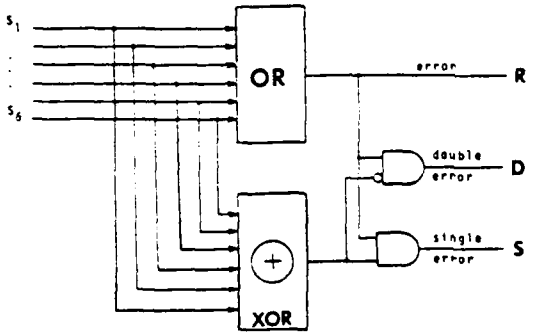


FIG. 10 The error detector.

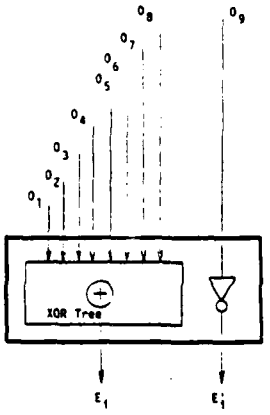


FIG. 11 Modified parity tree for the syndrome generator.

Finally, to distinguish between single and double errors, the E_i lines are input into a parity tree. Similarly, a second parity tree calculates the parity of the E'_i lines. Since the parity of the E_i lines (similarly E'_i lines) were made to be nonconstant, and since the E_i lines (similarly E'_i lines) satisfy Assumption A2, they can be made self-testing using Algorithm A. Figure 12 shows the self-testing embedded error detector portion of the SEC-DED circuit of Fig. 8. If all the inputs to the SEC-DED circuit are correct, then the syndrome generator produces a two-rail code and <R,R'> results in a 1-out-of-2 code. If one or more input lines are erroneous, then the output of the syndrome generator will not be two-rail, and hence <R,R'> does not form a 1-out-of-2 code. By the special encoding of [Hsiao 70] that is used here, and by the special design of the syndrome

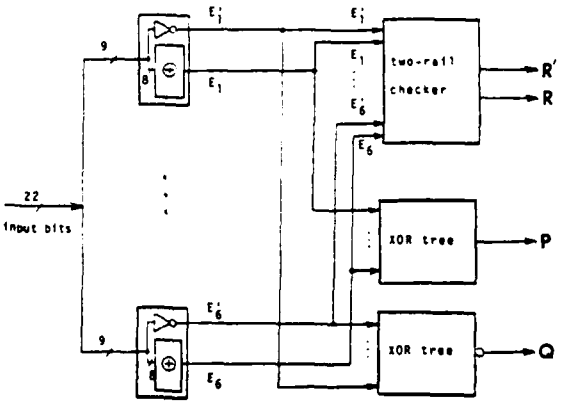


FIG. 12 Modified detector portion for self-testing embedded SEC-DED decoder circuit.

generator, one erroneous input results in an odd number of erroneous lines at the output of the syndrome generator, i.e., of the 12 output lines of the syndrome generator, an odd number would be erroneous. Similarly, if two input lines are erroneous, an even number of the output lines of the syndrome generator will be erroneous. It is not hard to see that the <P,Q> pair of Fig. 12 forms a 1-out-of-2 code if and only if there are an even number of errors on the 12 input lines to the two parity trees. This argument leads to Table 3, which indicates how to interpret the outputs of the circuit of Fig. 12.

R	R'	P	Q	meaning
0	1	-	-	correct input
1	0	-	-	
1	1	a	a	single error
0	0	a	a	
1	1	a	a'	double error
0	0	a	a'	

- a is 0 or 1.

TABLE 3 Reading of outputs of the circuit of Fig. 12.

CONCLUSION

A procedure has been developed for designing embedded parity checkers that are self-testing for all single stuck-at faults at the terminals of the XOR gates. This procedure has no hardware cost or speed degradation associated with it. However, applications of it to other parity-related code checkers may have a slight speed penalty (e.g., Fig. 11).

There is much room for expanding the ideas and methods presented in this paper. In particular (1) work needs to be done on finding other codes for which self-testing embedded checkers can be designed, and (2) other algorithms should be developed for detecting a more extensive set of faults in the checker. One such algorithm has been developed recently that results in an embedded parity tree that is self-testing for all faults within any single XOR gate [Khakbaz 82b].

ACKNOWLEDGMENTS

This work was supported by the U.S. Army Electronics Research and Development Command under Contract No. DAAK-20-80-0266. Many thanks to Professor E.J. McCluskey for his support and guidance.

REFERENCES

[Anderson 71] Anderson, D.A., "Design of self-checking digital networks using coding techniques," Tech. Rpt. R-527, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, 1971.

[Bossen 70] Bossen, D.C., D.L. Ostapko, and A.M. Patel, "Optimum test patterns for parity networks," PROC., AFIPS 1970 FALL JOINT COMPUTER CONFERENCE, Vol. 37, pp. 63-68, Houston, Texas, November 17-19, 1970.

[Carter 68] Carter, W.C. and P.R. Schneider, "Design of dynamically checked computers," PROC. 4TH CONGRESS IFIP, Vol. 2, pp. 878-883, Edinburgh, Scotland, August 5-10, 1968.

[Hsiao 70] Hsiao, M.Y., "A class of optimal minimum odd-weight-column SEC-DED codes," IBM J. OF RES. AND DEV., Vol. 14, No. 4, pp. 395-401, July 1970.

[Khakbaz 82a] Khakbaz, J., "Self-testing embedded parity trees," Technical Report, Computer Systems Laboratory, Stanford University, Stanford, Ca. 94305. To be published.

[Khakbaz 82b] Khakbaz, J., "Self-testing embedded parity trees - exhaustive XOR gate testing," Technical Report, Computer Systems Laboratory, Stanford University, Stanford, Ca. 94305. To be published.

[Smith 76] Smith, J.E., "The design of totally self-checking combinational circuits," Tech. Rpt. R-737, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, 1976.

[Wakerly 78] Wakerly, J.F., ERROR DETECTION CODES, SELF-CHECKING CIRCUITS AND APPLICATIONS, Elsevier North-Holland, Inc., New York, New York, 1978.

APPENDIX

Consider a set V of m-bit binary vectors. If v is in V, the set obtained by removing v from V is denoted by V-v. Also, the set obtained by adding a new vector w to V is denoted by V+w. Let p(V) be a vector obtained by bit-by-bit XORing of the vectors in V. That is, the ith element in p(V) is the parity of the ith row of a matrix whose columns are the members of V. Call p(V) the parity vector corresponding to V. Similarly, define p(V-v), p(V+w), and so on.

LEMMA 1 Let u, v, and w be m-bit binary column vectors. If both $u \oplus v$ and $u \oplus w$ are constant vectors, then either v and w are identical or they are complements of each other.

LEMMA 2 Let V be a set of m-bit binary column vectors. Let v be in V. Then, $p(V) = p(V-v) \oplus v$.

LEMMA 3 Let V be a set of m-bit binary column vectors. Let w be an m-bit binary column vector not in V. Then, $p(V+w) = p(V) \oplus w$.

The proofs of the above Lemmas are simple and directly follow the definitions.

LEMMA 4 If Assumptions A1 and A2 hold for the parity checker C, as exemplified in Fig. 3, then Algorithm A makes all the lines in C active.

PROOF First, show that after Step 1, both E1 and E2 are active. If they are active from the beginning, Step 1 does nothing, and the assertion is trivially true. So, assume E1 is passive at the beginning. Since the normal column corresponding to E1 is complementary to that of E2, E2 would also be passive. Thus p(S(E1)) and p(S(E2)) are constant vectors. The exchange in Step 1 results in two new sets of input lines corresponding to E1 and E2, as follows:

$$S'(E1) = S(E1) - Oi + Oj;$$
and $$S'(E2) = S(E2) - Oj + Oi.$$

But originally

$$S(E1) = S(E1) - Oi + Oi;$$
and $$S(E2) = S(E2) - Oj + Oj.$$

If p(S'(E1)) is also constant, then by Lemmas 1, 2, and 3 it is concluded that Oi and Oj are identical or complementary, contradicting Assumption A2. Similarly, it can be shown that p(S'(E2)) is not constant. Thus E1 and E2 are active at the end of Step 1.

Now consider Step 3. Let a and b be the two inputs to a gate whose output has been marked, but whose inputs have not been marked. If both a and b are active, they are marked. If, say, a is passive, exchange Oi in S(a) with Oj in S(b) to get:

$$S'(a) = S(a) - Oi + Oj; \quad (1)$$
and $$S'(b) = S(b) - Oj + Oi. \quad (2)$$

By a similar argument as above, it can be shown that this exchange makes a active. However, now b may be passive. In this case the specified exchange results in

$$S''(a) = S'(a) - Ok + Oi; \quad (3)$$
and $$S''(b) = S'(b) - Oi + Ok. \quad (4)$$

Existence of such Ok different from Oj in S'(a) is guaranteed, since otherwise S'(a) and hence S(a) would have only one member, which by Assumption A2 would imply that in fact a could not have been passive to start with. Substituting (1) in (3) and (2) in (4):

$$S''(a) = S(a) - Ok + Oj; \quad (5)$$
and $$S''(b) = S(b) - Oj + Ok. \quad (6)$$

Since S(a)=S(a)-Ok+Ok, and p(S(a)) is assumed to be constant (since a was originally passive), then p(S''(a)) may not be constant; otherwise (5) and the above Lemmas would yield that Ok is identical or complementary to Oj. Also since it was assumed that p(S'(b)) was constant (i.e, since it was assumed that the first exchange between Oi and Oj made b passive), then p(S''(b)) would not be constant; otherwise, (2) and (6) would imply that Oi is either identical or complementary to Ok. Therefore in at most two exchanges in Step 3 lines a and b become active and are subsequently marked. Since each time that Step 3 is executed two lines are marked, the Algorithm stops in a time proportional to the number of the lines in the tree. Finally, if t is the output line of the gate with input lines a and b, then, by the structure of the tree, S(t) = S(a) UNION S(b). Thus, any exchanges between S(a) and S(b) do not affect the fact that t (or, for that matter, any ancestor of t) is an active line. In other words, during the process of the Algorithm, all the marked lines remain active. Q.E.D.

THEOREM If Assumptions A1 and A2 hold for the parity checker C, then Algorithm A makes C self-testing for all single stuck-at faults at the terminal nodes of the XOR gates.

PROOF By Lemma 4, the Algorithm A makes all the lines in C active. Suppose line x in C is stuck at u (u is 0 or 1). Since x is active, there is a normal input pattern to C that, under fault-free condition, puts logic value u' on x. Assume x is in S(E1). Then, with x stuck at u, the above input pattern causes erroneous logic value on E1. Thus, <E1,E2> does not form a 1-out-of-2 codeword. Q.E.D.

WATCHDOG PROCESSORS AND CAPABILITY CHECKING

Masood Namjoo and Edward J. McCluskey

CENTER FOR RELIABLE COMPUTING
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, Ca. 94305

ABSTRACT

Applications of watchdog processors for detection of system malfunctions are described. Low-cost watchdog processors can be designed so that they have knowledge about the design specifications of a system and therefore can detect a large class of malfunctions by monitoring the run-time behavior of that system.

The concept of capability checking is introduced. Capability checking is aimed at the detection of malfunctions that cause illegal access to the memory system. It is shown that only a subset of such malfunctions is detected by the operating system. In the capability checking technique all access-right information is given in advance to an auxiliary low-cost processor, called a capability processor. The capability processor checks the validity of each access to the memory system dynamically. The implementation details of a capability processor are explained.

INTRODUCTION

One of the most basic techniques for checking the behavior of a system is the use of a watchdog timer [Con 72], [Orn 75]. The system is designed such that under normal operation it signals the watchdog timer within a specified time interval. This signal presets the timer to its initial value. The timer generates an error if no preset signal is received during that specified time interval. It is obvious that many malfunctions can occur while the system still generates a correct timing signal. In this paper we study the design of low-cost, yet more sophisticated watchdog processors for concurrent testing of a system. Watchdog processors [Lu 80] can be designed to have more knowledge about the design specifications of a system and hence be able to detect abnormal behaviors of that system at runtime.

Several approaches [Bol 78], [Yao 80] have been proposed for detection of malfunctions which result in control flow errors. Equally important is to study the effect of these malfunctions on the way the memory is referenced. This would be an attempt to detect system malfunctions as well as to prevent memory mutilation.

The idea of capability checking presented here is to use an auxiliary low-cost processor, called a Capability Processor (CP), to verify the validity of memory references. A typical configuration for the system is shown in Fig. 1. The capability processor operates in parallel with the CPU and detects a large class of illegal accesses to the memory system; a subset of these illegal accesses also is detected by the operating system.

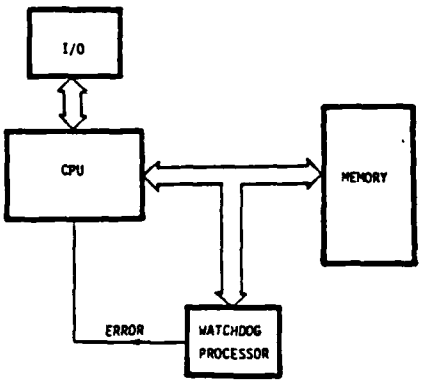


Figure 1

SYSTEM LEVEL MALFUNCTIONS

Classical methods of testing concentrate on functional testing at the circuit level. Unfortunately there exists a gap between the effect of faults at the circuit level and their behavior at the system level. As a very simple example consider a memory system that uses extra check bits for error detection and correction. Some multiple bit errors may go undetected at the circuit level. At the system level this may correspond to changing a correct instruction to an incorrect one causing the program to perform a different operation. We can detect such errors if:

1) The design specifications (from which the behavior of the system can be predicted) are known.

2) The errors cause abnormal behavior of the system.

Much research has been done in the area of operating systems which support protection [Sal 75]. In a typical descriptor-based system such as the IBM S/370 or the PDP-11/45 [Sal 75] the operating system loads the descriptor register with the base, limit, and the access right information. On the other hand in a capability-based system [Fab 74] such as the PLESSY S/250 [Eng 74] or the Cambridge CAP computer [Wil 79] the users themselves can load the descriptor register but only from a limited set of descriptor values (or capabilities) that has been given to them by the operating system.

Information used for the purpose of protection is stored in the memory, and in general, all protection systems assume fault-free hardware. This assumption, however, can be invalidated and protection violations can go undetected. This problem becomes more serious in virtual memory systems or capability-based systems where many page tables or capability-lists are stored in the main memory. There are three categories of errors that may not be detected by the operating system:

1) Errors in a memory word, protection registers, address bus, etc. caused by hardware failure.

2) A software error (accidental or malicious) in a user program.

3) A software error in a system routine which is assumed to be highly trustable.

As an example of a hardware failure that can result in protection violations, consider the paging system in the VAX-11 [Lev 80]. A program references the memory by giving the Virtual Page Number (VPN) and an offset in that page. The VPN points to an entry of the Process Page Table (PPT). A Process Base Register (PBR) points to the PPT. The physical address is formed by concatenation of the Page Frame Number (PFN), derived from the Page Table Entry (PTE), and the offset in the instruction, as shown in Fig. 2. The following hardware failures can cause a wrong memory access not detected by the protection system:

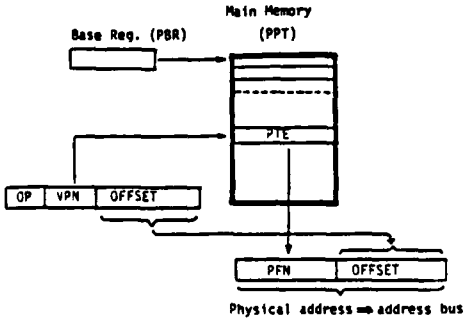


Figure 2

a. An error in the PBR or PTE (PBR and PTE can be changed only by the OS).

b. An error in the VPN. (part of the address in an instruction).

c. Failure of the access check mechanism (CPU failure) and invalid access attempt.

d. A fault on the address bus.

On the other hand, most software errors are due to design and coding errors and in general it is very difficult to guarantee that once the software passed its test, it is free of any errors [Yao 80].

CAPABILITY CHECKING

Before proceeding to the subject of capability checking it is helpful to define the terms which are used in this paper.

Definition: A system level malfunction is a deviation in the behavior of a system from its design specifications as a result of a hardware failure, a software error, or a design error.

In the presence of a system level malfunction the operation performed by the system is either illegal or incorrect.

Definition: An operation is illegal if, based on the design specifications, that operation is never allowed. For example execution from a "data" segment is an illegal operation.

Definition: An operation is incorrect if based on the design specifications and the current conditions, that operation is not correct. However the same operation can be correct under certain conditions.

For example if a program can write into two different data segments S1 and S2 depending on the value of a predicate "P", an incorrect operation would be to write into S2 instead of S1 as a result of an error in "P".

In general, detection of incorrect operations is more difficult than detection of illegal operations. Most incorrect operations occur as a result of incorrect decisions at branch points. These decisions in general can depend on the input data. Redundant predicates can be used to minimize the probability of a wrong decision [Kan 75]. In this paper we concentrate mostly on the detection of illegal operations, although some incorrect operations can also be detected.

Definition: An object is a set of logically contiguous memory cells whose type determines the class of operations that can be performed on it. A process can have a set of owned objects with full access to them. In addition, a process can be given access to some (external) objects by the owner of those objects. Examples of objects are: a program, a data segment, or a page.

Definition: A capability to an object is a special name that allows a specific access to that object. It has a unique logical address field, a type and an access right field.

At any given time a set O={O1,O2,...,On} represents the set of all active objects. Ci={Ci1,Ci2,...,Cik} is the set of capabilities that are given to the object Oi. An object Oi has to present a capability Cij in order to access the object Oj. The access right is aij. The operation of the capability processor is as follows:

A. From the point of view of the capability processor each process is defined by a set of code and data objects. The set of active objects (stored in the physical segments of the primary memory) can be represented by a directed graph; an example is shown in Fig. 3. A vertex in this graph represents an object. An edge shows the access right of an object to another.

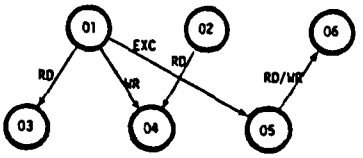


Figure 3

Before a program is initiated, all access-right information is sent to the CP. This is done by loading the Segment Access Table (SAT) and the Segment Map Table (SMT) in the CP. The row Si (Si is the segment ID of the object Oi) in the SAT contains the set of access rights for the object Oi (i.e. aij for j=1..k). An entry SAT(Si,Sj) in the SAT shows the access right of the object Oi to the object Oj. A null entry denotes the no access situation. For any code object Oi, the entry SAT(Si,Si) is an execute access right. An object Ok can be shared between two code objects Oi and Oj with different access rights:

SAT(Si,Sk)=aik and SAT(Sj,Sk)=ajk ;aik≠ajk

B. For each memory reference, the physical address is translated to a segment ID using the SMT. This segment ID is used in turn as the address for accessing the SAT. Two segment IDs are required to access the SAT. The first is the segment ID (Si) of the current code object (Oi). The second is the segment ID (Sj) of the object (Oj) referenced by the current object. Si and Sj are determined from the physical address in each reference and the mapping information in the SMT. If the requested access by the CPU is not consistent with SAT(Si,Sj) which is read out from the SAT, the capability processor signals the main processor and the main processor initiates a recovery routine for handling the detected error.

IMPLEMENTATION

The following assumptions are made: First, all access information given to the capability processor by the main processor is error free. If not, the CP may signal an access error while an access is legal and fail to signal while the access is illegal. Second, the probability of simultaneous failure of the main system and the capability processor is very low. Third, all accesses from any location in an object Ox to any location in another object Oy are "equivalent". In other words if the object Ox can write into the object Oy, this technique would not check whether or not the referenced location within Oy is correct.

The first stage of the capability processor is an address translator. It translates a physical address into a segment ID using the mapping information, and it determines the type of the segment (code or data). In the case of a paged memory system all references to different pages of an object are mapped onto a unique segment ID for that object using the SMT. This requires one access to the SMT.

In Fig. 4 register Rx holds the segment ID of the current code segment, Si, which is determined from the current memory reference using the mapping data in the SMT. The segment ID for the next reference to the memory, Sj, is also determined and loaded into register Ry by the capability processor. The entry SAT(Si,Sj) is read out from the SAT and is compared with the accesa requested by the CPU. The watchdog signals an error if this comparison fails. Notice that in this method the capability processor checks the validity of each access in parallel with the CPU operation. Once a successful access is completed, the capability processor loads Rx from Ry only if Ry holds the segment ID of a code segment. This operation is repeated for each memory reference.

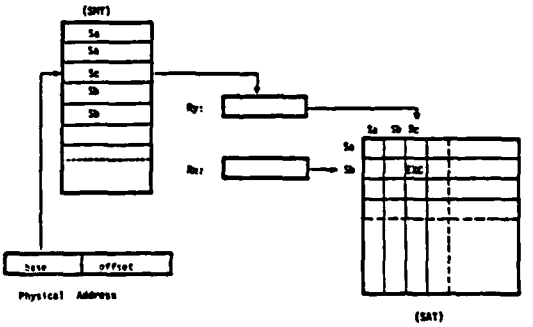


Figure 4

Operations are divided into three classes: the first class includes operations in which an operand is accessed by the operation. Examples are read, write, add, and move. In this case the instruction is fetched from a code segment and the operand is in a data segment. The content of Rx will not change after such operations.

The second class includes simple control operations such as jump and set flag. The expected access right for such operations is "execute" and the content of Rx will change after such operations if and only if a transfer to another segment occurs.

The third class includes call and return operations. In this case the expected access right is "enter" or "return" and again the content of Rx will change after such operations if and only if a transfer to another segment occurs.

Capability checking can be implemented using memories that are as fast as the main memory. When the main processor loads the pages of a process into the main memory, the corresponding capabilities are loaded into the SAT. During the execution of a process, some of its pages may be swapped in and out of the physical memory. For each page replacement, only one entry in the SMT is updated. The entries of the SMT corresponding to the removed pages are marked as invalid and any access to these pages is considered illegal.

In capability checking the accessibility of memory segments is checked on the basis of physical addresses at the processor-memory interface. Once an illegal access is detected the CPU is informed and a recovery routine is initiated. Since the CP operates in parallel with the CPU, it does not degrade the system performance. Notice that updating the SAT and the SMT can be overlapped with the time required for swap-in and swap-out of the pages which is a slow operation. It is also possible to take samples of the memory references (at a slower rate) and use the same concept for checking the capabilities for those samples. However, in this case since the checking is not done exhaustively, some illegal accesses may go undetected.

CONCLUSIONS

Low-cost watchdog processors can be designed to detect abnormal behaviors of a system under operation. In capability checking the accessibility of each memory reference is checked on the basis of physical addresses at the processor-memory interface. Since the checking is done in parallel with the main processor, there is no degradation in the system performance. However, there is a possibility that a few illegal accesses occur before the CP signals the main processor. In order to keep track of illegally accessed locations, a buffer can be used to save the address of the last m (e.g. m=10) references.

Such a capability processor can be used as a redundant protection scheme in systems where a high degree of security is required. On the other hand, this method by itself is an economical way for increasing the reliability of small systems.

Current active research in this area includes the design of watchdog processors for checking the flow of execution or the integrity of data structures. Preliminary results in this area are given in [Nam 81].

ACKNOWLEDGMENTS

The authors wish to thank members of the Center for Reliable Computing at Stanford University. Helpful comments by David J. Lu and also the referees are greatly appreciated. This work was supported in part by the US Army Electronics, Contract No. DAAK20-80-C-0266.

REFERENCES

[Bol 78] S. Bologna and W. Ehrenberger, "Possibilities and Boundaries for the Use of Control Sequence Checking," Proc. of the Eight Annual International Conference on Fault-Tolerant Computing, IEEE, June 1978, p.226.

[Con 72] J. R. Connet, E. J. Pasternak, and B. D. Wagner, "Software Defenses in Real-Time Control Systems," Digest 1972 Int. Symp. on Fault-Tolerant Computing, June 1972, pp. 94-99.

[Eng 74] D. M. England, "Capability Concept Mechanisms and Structure in System 250," IRIA Int. Workshop on Protection in Operating Systems, August 1974, pp. 63-82.

[Fab 74] R. S. Fabry, "Capability-Based Addressing," Comm. ACM, July 1974, pp. 403-412.

[Kan 75] J. R. Kane and S. S. Yao, "Concurrent Software Fault Detection," IEEE, Transaction on Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 87-99.

[Lev 80] H. M. Levy and R. H. Eckhouse,Jr., Computer Programing and Architecture, VAX-11, Digital press, Digital Equipment corporation, Bedford, Mass. 1980.

[Lu 80] D. J. Lu, "Watchdog processors and VLSI," Proc. of the National Electronics Conference, Vol. 34, Chicago, October 1980, pp. 240-245.

[Nam 81] M. Namjoo and E. J. McCluskey, "Watchdog Processors and Detection of Malfunctions at the System Level," CRC Rep. No. 81-17, December 1981, Stanford University, Stanford, Ca. 94305.

[Orn 75] S. M. Ornstein, W. P. Crowther, M. F. Kraley, R. D Bressler, A. Michel, and F. E. Heart, "Pluribus - A Reliable Multiprocessor," AFIPS Int. Computer Conf. 1975, pp. 551-559.

[Sal 75] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems" Proc. of the IEEE, Vol. 63, No. 9, September 1975, pp. 1278-1308.

[Wil 79] M. V. Wilkes and R. M. Needham, The Cambridge CAP Computer and its Operating System, Elsevier North Holland Inc., 1979.

[Yao 80] S. S. Yao, and F. Chen, "An Approach to Concurrent control Flow Checking," IEEE, Transaction on Software Engineering, Vol. SE-6, No.2, March 1980, pp. 126-137.

AUTONOMOUS LINEAR FEEDBACK SHIFT REGISTER WITH ON-LINF FAULT-DETECTION CAPABILITY

Laung-Terng Wang*

CENTER FOR RELIABLE COMPUTING, COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, California 94305

ABSTRACT

In this paper, an encoder algorithm for the design of an autonomous Linear Feedback Shift Register (LFSR) with specified minimum distance and cycle length is presented. The fault detectability on the feedback path of this LFSR encoder is then discussed. This shift register design significantly extends the work in the literature [Hsiao 77] [Pradhan 78], and is based on cyclic codes.

1 INTRODUCTION

An autonomous Linear Feedback Shift Register (LFSR) is an autonomous linear sequential network [Elspas 59] [Kautz 65] [Zierler 59] for generating sequences of a given cycle length (or period). This LFSR is composed of interconnections of unit-delays (or D Flip-Flops) and modulo-2 adders (or Exclusive OR gates), as shown in Figure 1.

The LFSR has been used in many different applications. Example of these applications are: pseudo-random number generators [Golomb 67], signature analyzers [Benowitz 75] [McCluskey 81], shift register counters [Gschwind 75], store address generators [Hsiao 77] [Pradhan 78], etc. For instance, in the LSI/VLSI chip designs using a random testing scheme [Losq 76], the random input sequence feeding into the Device Under Test (DUT) can be autonomously generated by an LFSR of minimum (Hamming) distance 1, or by an LFSR encoder of minimum distance at least 2 [Hsiao 77].

An LFSR of minimum distance 1 (or distance-1 LFSR) is an autonomous LFSR with minimum Hamming distance 1 among the generated states. It cannot detect any fault inside itself. If a fault (or error) occurs that causes a faulty input sequence to the DUT, albeit good, the output response will be incorrect which may result in the DUT being rejected. This fault may be detectable on-line if an LFSR encoder of minimum distance at least 2 followed by an error detector is adopted. It depends on how big the minimum distance is used.

This work was supported in part by the National Science Foundation under the Grant Number MCS-7904864, and in part by Intel Corporation under the Honors Cooperative Program (HCP).

*The author is also with Intel Corporation, Santa Clara, CA 95051.

2 LFSR PROPERTIES

Figure 1 shows a general form of an n-stage LFSR with corresponding characteristic polynomial, defined by

$$f(x) = 1+h_1x+h_2x^2+\ldots+h_{n-1}x^{n-1}+x^n, \qquad (1)$$

where h_i $(1 \le i \le n-1)$ is either one or zero.

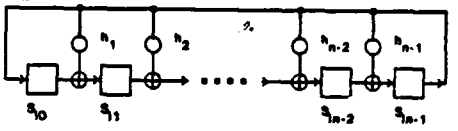


Figure 1. The general form of an LFSR.

The behavior of an LFSR can be interpreted as an ordered cyclic chain of states S; which are symbolic representation of the contents of an LFSR during successive shifts, given the initial contents as S_0. Let S_i represent the contents of an n-stage LFSR after the i^{th} shift of the initial contents, S_0, of the LFSR, and $S_i(x)$ be the polynomial representation of S_i, then $S_i(x)$ is a polynomial of degree n-1,

$$S_i(x) = S_{i0}+S_{i1}x+\ldots+S_{in-1}x^{n-1}. \qquad (2)$$

The following is a fundamental relationship between the states in a cycle [Hsiao 77]:

$$S_i(x) = x^{i-j}S_j(x) \bmod f(x). \qquad (3)$$

If T is the least positive integer such that f(x) divides x^T-1, then for any state $S_i(x)$,

$$S_i(x) = x^T S_i(x) \bmod f(x). \qquad (4)$$

The integer T is called the exponent of f(x) and the period of the LFSR.

3 ENCODER DESIGN

Let a polynomial with coefficients in the Galois Field GF(q) [Peterson 72] be said to be a polynomial over GF(q). A polynomial p(x) of degree m over GF(q) is called primitive if its root b of

$GF(q^m)$ with $p(b)=0$ generates all the nonzero elements of $GF(q^m)$. A polynomial $g(x)$ of degree $n-k$ over $GF(q)$ for generating an (n,k) cyclic code is called a generator polynomial if it is unique and is a divisor of x^n-1. An (n,k) cyclic code is an (n,k) linear code containing a set of q^k n-tuple distinct code words with the following property: If an n-tuple is a code word, then the resulting n-tuple by rotating the code word one place to the right is also a code word [Lin 70].

Let $p(x)$ and $g(x)$ denote a primitive polynomial of degree k and a generator polynomial of degree $n-k$ over Galois Field $GF(q)$, respectively. [Hsiao 77] has shown that given a required k message digits and a desired design (minimum) distance d_{min}, an autonomous LFSR can be constructed to generate q^k-1 n-tuple distinct code words by using the characteristic polynomial $f(x)=g(x)p(x)$. The initial contents, $S_0(x)$, of the LFSR can be preset to any nonzero code word. This theorem is based on an (n,k) cyclic code with $g(x)$ dividing x^n-1, and is applicable to an LFSR design with period $T=q^{k-s}-1$ for $0<s<k$ by deleting s digits from the k message digits. The result is an $(n-s,k-s)$ shortened cyclic code. However, it is not applicable to an LFSR generating an arbitrary period. With period T not equal to $q^{k-s}-1$, the initial contents of the LFSR have to be chosen very carefully to avoid producing an incorrect period.

Definition 1: An (n,k,T) LFSR encoder is an n-stage autonomous LFSR for generating T n-tuple distinct code words (or states) by the T consecutive shifts of the LFSR. It consists of k message digits and $n-k$ parity check digits.

Theorem 1: An (n,k,T) LFSR encoder can be constructed using $f(x)=g(x)p(x)$ as a characteristic polynomial over $GF(q)$, if the initial contents, $S_0(x)$, of the LFSR is divisible by $g(x)$, i.e., $S_0(x)=g(x)a(x)$, and both $a(x)$ and $p(x)$ have no common factor, where $g(x)$ is a generator polynomial of degree $r=n-k$ for generating an (n,k) cyclic code, $p(x)$ is a polynomial of the smallest degree k for generating a prescribed period T, and $a(x)$ is a polynomial of degree $k-1$ or less.

Proof: See [Wang 82].

Theorem 1 is applicable to an $(n-s,k-s,T')$ LFSR encoder design by deleting s digits from the k message digits for $0<s<k$. It implies that every code word $S_i(x)$ of an LFSR encoder is divisible by $g(x)$, and the Greatest Common Divisor (GCD) of $S_0(x)$ and $p(x)$ is $g(x)$, i.e., $GCD(S_0(x), p(x)) = g(x)$. If the GCD of $S_0(x)$ and $p(x)$ is not $g(x)$, the LFSR will produce an incorrect period and may result in different minimum distance.

Example 1: Design a $(6,3,4)$ LFSR encoder using $f(x)=g(x)p(x)=(1+x+x^3)*[(1+x)(1+x^2)]=1+x^3+x^5+x^6$. The circuit is shown in Figure 2. The desired $S_0(x)$'s for the $(6,3,4)$ LFSR encoder are marked by (*). Each $S_0(x)$ and $p(x)$ have a GCD $g(x)=1+x+x^3$.

Table 1. The $S_0(x)$'s with resulting T and d_{min}.

$S_0(x)$	T	d_{min}
$1+x+x^3$	4	4 (*)
$(1+x+x^3)(1+x)$	2	4
$(1+x+x^3)(1+x^2)$	1	0
$(1+x+x^3)(1+x+x^2)$	4	4 (*)

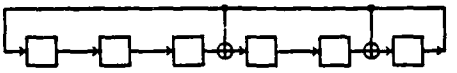


Figure 2. A (6,3,4) LFSR encoder

The synthesis of a polynomial $p(x)$ of the smallest degree k for a prescribed period T was presented in [Wang 82]. Theorem 2 provides an encoder algorithm to derive the required $g(x)$ based on Bose-Chaudhuri-Hocquenghem (BCH) codes [Peterson 72]. BCH codes are cyclic codes. Let b be an element of $GF(q^m)$. For any specified integer c and design distance d, the code generated by $g(x)$ is a BCH code, if and only if $g(x)$ is the polynomial of the smallest degree over $GF(q)$ for which b^c, b^{c+1},..., b^{c+d-2} are roots.

Theorem 2: A $(2^m-1,k,T)$ LFSR encoder with design distance, d_{min}, at least $2t+1$, where t is an integer, can be constructed using $f(x)=g(x)p(x)$ as a characteristic polynomial, if the polynomial $p(x)$ is of the smallest degree k for generating a prescribed period T, and the generator polynomial $g(x)$ of the code is given by

$$g(x) = LCM<m_1(x), m_3(x),..., m_{2t-1}(x)>, \qquad (5)$$

the Least Common Multiple (LCM) of $m_i(x)$'s of degree m $(i=1,3,5,...,2t-1)$, where $m_1(x)$ is a primitive polynomial of degree m, its root b over $GF(2^m)$ is of order 2^m-1, and $m_i(x)(i=3,5,...,2t-1)$ is the minimum polynomial of b^i.

Proof: See [Wang 82].

A minimum polynomial $m_i(x)$ of root b^i over $GF(q^m)$ is a polynomial of the smallest degree over $GF(q)$ such that $m_i(b^i)=0$. Table C.2 [Peterson 72] provides a list of minimum polynomials of root b^i of degree 34 or less. Since a primitive polynomial is a minimum polynomial of root b, $g(x)$ can thus be found from Table C.2 [Peterson 72]. Table 2 lists the required $g(x)$'s of degree r for designing some of $(2^m-1,k,T)$ LFSR encoders with $d_{min}=3$, 5, or 7.

The number of k message digits was obtained by setting it to 2^m-1-r.

Table 2. The g(x)'s for $(2^m-1,k,T)$ LFSR encoders of $d_{min}=3$, 5, or 7.

$(2^m-1,k,T)$	d_{min}	g(x)
(7,4,5)	3	$1+x+x^3$
(15,11,23)	3	$1+x+x^4$
(15,7,127)	5	$(1+x+x^4)(1+x+x^2+x^3+x^4)$
(15,5,31)	7	$(1+x+x^4)(1+x+x^2+x^3+x^4)(1+x+x^2)$

The above theorem is applicable to $(n',k',T') = (2^m-1-s,k-s,T')$ LFSR encoders [Hsiao 77] [Lin 70] [Peterson 72] for $0<s<2^{m-1}$. Since f(x) always contains a factor $1+x$ when T is even, by the property [Hsiao 77] [Pradhan 78] [Peterson 72] that the minimum distance of the generated code space is even if $1+x$ is a divisor of f(x), the resulting design distance of the LFSR encoder will be an even number $2t+2$, if the period T is even; and an odd number $2t+1$, if T is odd. In implementing an LFSR encoder with desired even minimum distance of at lesst $2t+2$ for a prescribed odd period T, the generator polynomial should be modified as

$$g(x) = (1+x)*LCM<m_1(x), m_3(x),..., m_{2t-1}(x)>. \quad (6)$$

Example 2: The following examples were derived from Table 2 by deleting some message digits. For instance, the (6,3,4) was derived from the (7,4,5) of $d_{min}=3$ by deleting one message digit. Since the period 4 is an even number, the (6,3,4) LFSR encoder will have a design distance $3+1=4$.

Table 3. $(2^m-1-s,k-s,T')$ LFSR encoders.

(n',k',T')	d_{min}	g(x)	p(x)
(6,3,4)	4	$1+x+x^3$	$1+x+x^2+x^3$
(11,3,4)	6	$(1+x+x^4)(1+x+x^2+x^3+x^4)$	$1+x+x^2+x^3$
(7,3,7)	4	$(1+x)(1+x+x^3)$	$1+x+x^3$
(12,3,7)	6	$(1+x)(1+x+x^4)(1+x+x^2+x^3+x^4)$	$1+x+x^3$

4 FAULT DETECTABILITY

An encoder algorithm for the LFSR design with desired period T and minimum distance d_{min} has been presented. This LFSR encoder gives a Totally Self-Checking (TSC) error detector the on-line fault-detection capability to detect at most $d_{min}-1$ errors on the encoder output [Wang 82]. The fault model can be any combination of faults, such as stuck-at faults, bridging faults, and external disturbances (or noise), which manifest themselves by changing at most $d_{min}-1$ positions on the output of the LFSR encoder. However, it is not clear whether the fault that makes the entire feedback path assume an erroneous state is detectable or not. For instances, an error on the feedback path of Figure 2 may produce 3 errors in the state.

Definition 2: An $(n,k,T)=(2^m-1,k,2^k-1)$ cyclic code is an (n,k) cyclic code of period 2^k-1. An $(n',k',2^{k'}-1)=(2^m-1-s,k-s,2^{k-s}-1)$ shortened cyclic code is an (n',k') shortened cyclic code of period $2^{k-s}-1$, by deleting s digits from the k message digits for $0<s<2^{m-1}$.

[Hsiao 77] has shown that in the $(n,k,2^k-1) = (2^m-1,k,2^k-1)$ cyclic codes, the effect of a fault on the autonomous LFSR feedback path is only to produce a noncode word which differs in the first stage of the LFSR from some valid code word. In implementing an LFSR encoder by $(n',k',2^{k'}-1) = (2^m-1-s,k-s,2^{k-s}-1)$ shortened cyclic codes for $0<s<2^{m-1}$, the effect of a fault on the LFSR feedback path is proven to produce errors exactly at distance $d_{min}-1$ from some valid code word.

Theorem 3: In an $(n',k',2^{k'}-1)$ LFSR encoder design implementing a shortened cyclic code, the effect of a fault that makes the entire feedback path assume an erroneous value is to change the correct state (a code word) to an erroneous state (noncode word) which is exactly at distance $d_{min}-1$ from some other valid state.

Proof: See [Wang 82].

Theorem 3 extends the fault detectability in [Hsiao 77] to $(n',k',2^{k'}-1)$ shortened cyclic codes. For a code of period T' not equal to $2^{k'}-1$, the fault, making the entire feedback path assume an erroneous state in realizing an LFSR encoder, may certainly cause a noncode word at distance more than 1 for $n'=2^m-1$ (or at distance more than $d_{min}-1$ for n' not equal to 2^m-1) from the T' states. Fortunately, since the same g(x) can be used to realize both (n',k',T') and $(n',k',2^{k'}-1)$ LFSR encoders, and the same error detector can be adopted to detect errors within themselves, the fault detectability will be the same when both criteria are implied. Moreover, since the generator polynomial g(x) in an LFSR encoder of an even period of $d_{min}=2t+2$ is one degree less than that in an LFSR encoder of an odd period of $d_{min}=2t+1$ for t-error-correcting, the produced noncode word due to a fault on the feedback path will be always at distance 2t, irrelevant of the period being even or odd. This is summarized below:

Corollary 1: Suppose that the same generator polynomial g(x) is used to implement both (n',k',T') and $(n',k',2^{k'}-1)$ LFSR encoders for T' not equal to $2^{k'}-1$, and both LFSR encoders use the same error detector to detect errors within themselves. Then the effect of the fault, which makes the entire feedback path assume an erroneous state in realizing an LFSR encoder of period T', will produce a noncode word which (1) differs in the first stage of the LFSR encoder from some code

word in the $(n,k,2^k-1)$ cyclic codes, and (2) is exactly at distance $d_{min}-1$ (or $d_{min}-2$) from some code word, for T' odd (or even), in $(n',k',2^{k'}-1)$ shortened cyclic codes, respectively.

Example 3: A (6,3,4) distance-4 LFSR encoder can use the same error detector as a (6,3,7) distance-3 LFSR encoder. Figure 3 shows the (6,3,7) distance-3 LFSR encoder using $f(x)=g(x)p(x)=(1+x+x^3)*(1+x^2+x^3)$ $=1+x+x^2+x^3+x^4+x^5+x^6$. The code space consisting of the 7 6-tuple nonzero code words is given in Table 4. Marked by (*) are the states generated by a (6,3,4) distance-4 LFSR encoder as shown in Figure 2. If the feedback path on both (6,3,7) and (6,3,4) LFSR encoders were stuck at one, then the next noncode words for the same input code word <110100>, for instance, would be <100101> and <111111>, respectively. Both invalid code words are exactly at (minimum) distance 2 from the code words <100011> and <101110>, respectively, although they produce up to 6 and 3 errors on the next states, respectively. Thus, the fault can be immediately detected by the error detector.

Table 4. Code space of a (6,3,7) distance-3 LFSR encoder.

1	1	0	1	0	0	(*)
0	1	1	0	1	0	(*)
0	0	1	1	0	1	(*)
1	1	1	0	0	1	
1	0	0	0	1	1	(*)
1	0	1	1	1	0	
0	1	0	1	1	1	

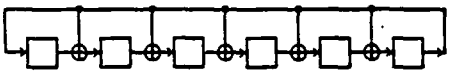


Figure 3. A (6,3,7) distance-3 LFSR encoder.

5 CONCLUSIONS

In this paper, an encoder algorithm is presented to derive the required characteristic polynomial and the initial contents for an LFSR encoder design with specified minimum distance and cycle length. It shows that in designing an LFSR encoder with cycle length not equal to 2^k-1, for k an integer, its initial contents have to be chosen under certain circumstances to avoid resulting in a shortened cycle length. Any LFSR encoder with even cycle length always produces an even design (minimum) distance. The fault detectability study on the feedback path of the LFSR encoder implementing shortened cyclic codes indicates that the fault, making the entire feedback path assume an erroneous state, is to produce a noncode word exactly at distance $d_{min}-1$ from some code word. For any combination of faults, such as stuck-at faults, bridging faults, and external disturbances (or noise) within the circuit, which manifest themselves by changing at most $d_{min}-1$ positions on the LFSR encoder output, the faults are detectable.

6 ACKNOWLEDGMENTS

The author wishes to thank Professor E. J. McCluskey and Dr. D. J. Lu for their guidance and instruction during the research work. Special thanks also go to K. Hose, D. Tjoa, S. Hassan, H. Dong and A. Mahmood for their constructive suggestions. The help from L. Christopher and H. J. Chang in preparing this paper is also deeply appreciated. This research work was supported in part by the National Science Foundation under Grant Number MCS-7904864, and in part by Intel Corporation under the Honors Cooperative Program (HCP).

7 REFERENCES

[Benowitz 75] Benowitz, N., D.F. Calhoun, G.E. Alderson, J.E. Bauer, and C.T. Joeckel, "An Advanced Fault Isolation System for Digital Logic," IEEE Trans. on Computers, Vol. C-24, No. 5, pp. 489-497, May 1975.

[Elspas 59] Elspas, B., "The Theory of Autonomous Linear Sequential Networks," IRE Trans. on Circuit Theory, Vol. CT-6, No. 1, pp. 45-60, March 1959.

[Golomb 67] Golomb, S.W., Shift Register Sequences, Holden-Day Inc., San Francisco, California, 1967.

[Gschwind 75] Gschwind, H.W., and E.J. McCluskey, Design of Digital Computers, 2nd edition, New York: Springer, 1975.

[Hsiao 77] Hsiao, M.Y., A.M. Patel, and D.K. Pradhan, "Store Address Generator with On-Line fault-Detection Capability," IEEE Trans. on Computers, Vol. C-26, No. 11, pp. 1144-1147, November 1977.

[Kautz 65] Kautz, W.H., editor, Linear Sequential Sitching Circuits, Holden-Day Inc., San Francisco, California, 1965.

[Lin 70] Lin, S., An Introduction to Error-Correcting Codes, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1970.

[Losq 76] Losq, J., "Referenceless Random Testing," Proc. 6th Annual Symposium on Fault-Tolerant Computing (FTCS-6), pp. 108-113, Pittsburgh, Pennsylvania, June 1976.

[McCluskey 81] McCluskey, E.J., and S. Bozorgui-Nesbat, "Design for Autonomous Test," IEEE Trans. on Circuits and Systems, Vol. CAS-28, No.11 pp. 1070-1079, November 1981.

[Peterson 72] Peterson, W.W., and E.J. Weldon, Jr., Error Correcting Codes, 2nd edition, M.I.T. Press, Cambridge, Massachusetts and London, 1972.

[Pradhan 78] Pradhan, D.K., M.Y. Hsiao, A.M. Patel, and S.Y. Su, "Shift Registers Desigred for On-Line Fault Detection," Proc. 8th Annual Symposium on Fault-Tolerant Computing (FTCS-8), pp. 173-178, Toulouse, France, June 1978.

[Wang 82] Wang, L.T., "The Encoder and Totally Self-Checking Error Detector Design of Autonomous Linear Feedback Shift Registers," Center for Reliable Comp. (CRC) Technical Report, Computer Systems Lab., Stanford University, May 1982.

[Zierler 59] Zierler, N., "Linear Recurring Sequences," Society for Industrial and Applied Mathematics Journal, Vol. 7, No. 1, pp. 31-48, Philadelphia, March 1959.


DATE
FILMED
7-8